プログラマの為の量子コンピュータ入門

Part 2:
量子ゲート型のプログラミング

先端IT活用推進コンソーシアム
オープンラボ

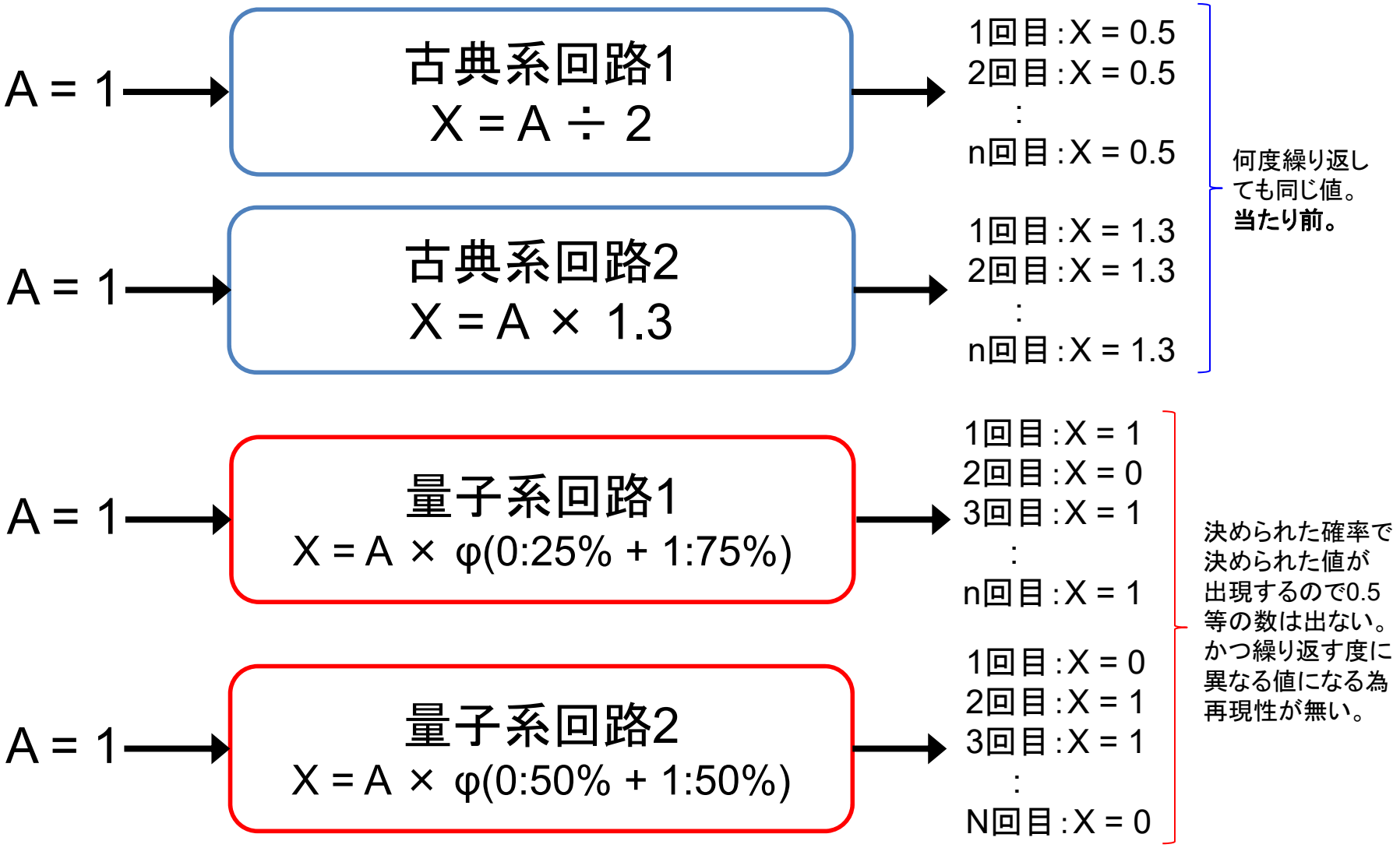
宮地直人（miyachi@langedge.jp）

Ver1.0 2019年8月26日

Part 0：イントロダクション

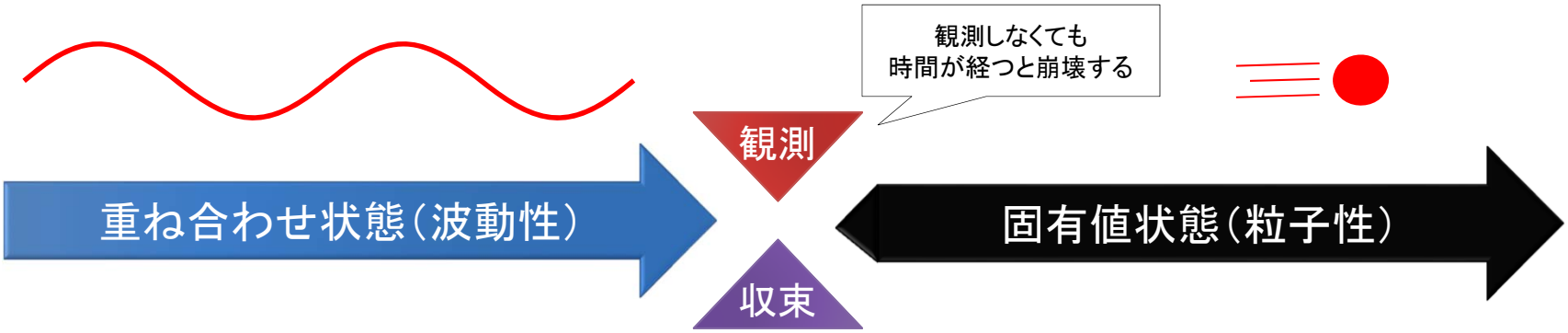
（振り返り短縮版）

古典系と量子系の測定



φ : 波動関数（複数の解を持つ）

波動と粒子の二重性（量子重ね合わせ）



複数の状態が同時に存在して
量子干渉を生じている
（コペンハーゲン解釈）

波動関数（列ベクトル）を
$|\psi>$ と表現すると

$$|\psi> = c_0|\psi_0> + c_1|\psi_1>$$
$$|c_0|^2 + |c_1|^2 = 1.0$$

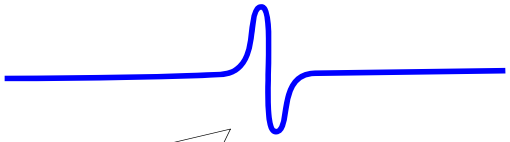
c_0 と c_1 は複数数

量子デコヒーレンス

固有状態のうち1つの状態

$|\psi_0\rangle$ または $|\psi_1\rangle$ のいずれか

波動性は失われる
複素数虚数部は意味を持たない



コペンハーゲン解釈では、粒子状態
は波動がパルス状になったと考える。
位置が決まると波長は定まらなくなる。

※ コヒーレンス時間（状態の量子干渉が失われるまでの時間）は短い。

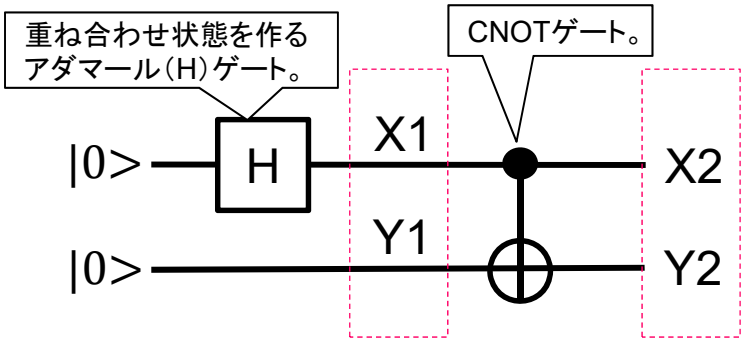
量子ビット（重ね合わせの実現）

- 0と1の量子重ね合わせの状態が可能な単位。
- 観測により2つの固有値（0か1）に収束する。
- 物理的に実現するには幾つかの方法がある。

方式	概要	開発
超伝導量子ビット	現在主流となっている超伝導状態のシリコン回路で量子ビットを実現する方式（極低温）	IBM, Google, D-Wave
イオントラップ	捕獲したイオンをレーザーで冷却して利用（室温） 理論的には量子ビット間の全結合が可能	IonQ
量子ドット	原子10～50個で構成した微小半導体を利用（極低温?）	Intel
トポロジカル	超電導体とトポロジカル絶縁体による量子ビット（極低温?）	Microsoft
NVセンター ダイヤモンド窒素-空孔中心	ダイヤモンドの炭素を窒素に置き換えて生じる欠損部に電子を捕獲して量子ビットに利用（室温）	（研究レベル）
光子 光量子コンピュータ	光子パルス群を量子ビットとして利用（室温） 全結合によるアニーリング型の計算が可能	NTT/NII/東大（ImPACT）

量子もつれ（量子エンタングルメント）

- 量子もつれは、2つの粒子（量子ビット）が互いに影響をおよぼし合い、一方を測定すると、もう一方の値が確定する現象である。
- 複数の量子ビット間を、量子もつれにより関連付けることで量子回路を構築する。以下HとCNOTによる例。



X2が0ならY2も0に、
Y2が1ならX2も1と、観測

$X1 = |0\rangle : 50\% + |1\rangle : 50\%$
$Y1 = |0\rangle$

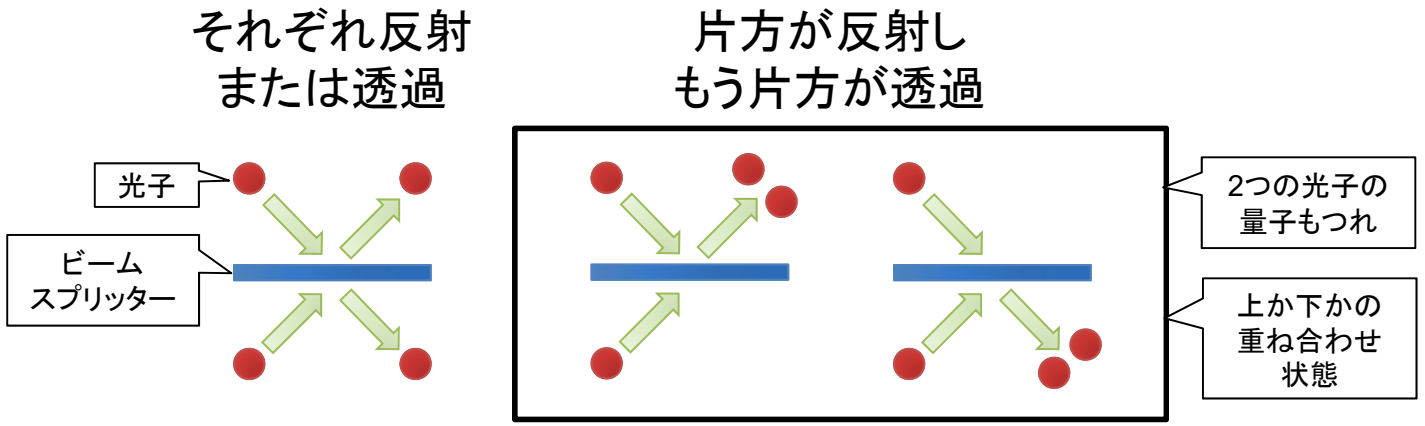
$X2 = X1 = |0\rangle : 50\% + |1\rangle : 50\%$
X1が $|0\rangle$ なら Y2も $|0\rangle$ （Y1そのまま）
X1が $|1\rangle$ なら Y2も $|1\rangle$ （Y1を反転する）

$X2Y2 = |00\rangle : 50\% + |11\rangle : 50\%$
※ $|01\rangle$ や $|10\rangle$ は 0%

光の量子コンピュータ（※ 前回未説明）

上下から光子対を入射する。

・ビームスプリッターとはハーフミラーのこと。

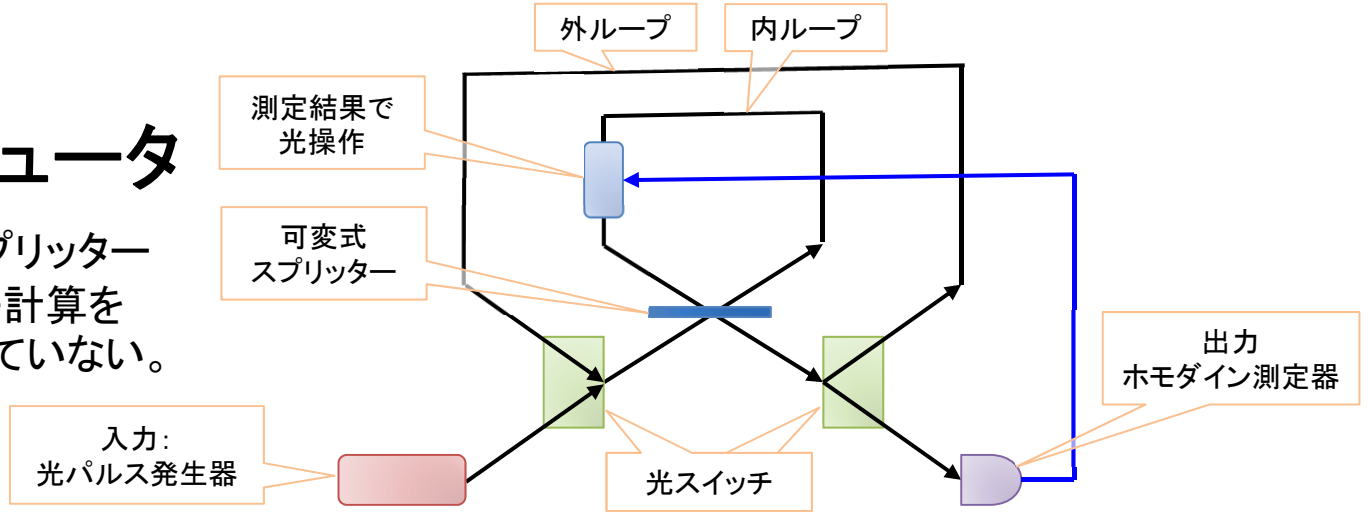


※ 測定するとこちらのいずれかになる

ループ型光量子コンピュータ

1つの可変式ビームスプリッターを何度も使うことで量子計算を行っていく。まだ完成していない。

複数のビームスプリッターを使うことで量子計算を行っていくタイプもある。



可逆コンピュータ

可逆コンピュータは非量子な場合には低電力になっても低性能になる為にほぼ実用化されていない。

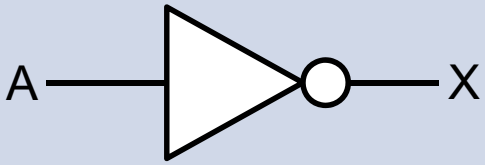
情報が失われる時にエネルギー（熱）が消費される。

可逆コンピュータが実現すると低電力化が可能となる。

IBMから1960年代に出た理論で今も研究が続いている。

※ 量子コンピュータは可逆コンピュータの一種である。

NOTゲート

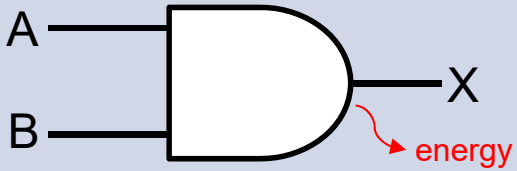


A	X
1	0
0	1

出力から入力が再現できるので可逆

可逆

ANDゲート

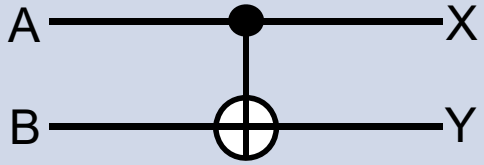


A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

出力から入力が再現できない

非可逆（量子計算では使えない）

CNOTゲート



A	B	X	Y
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

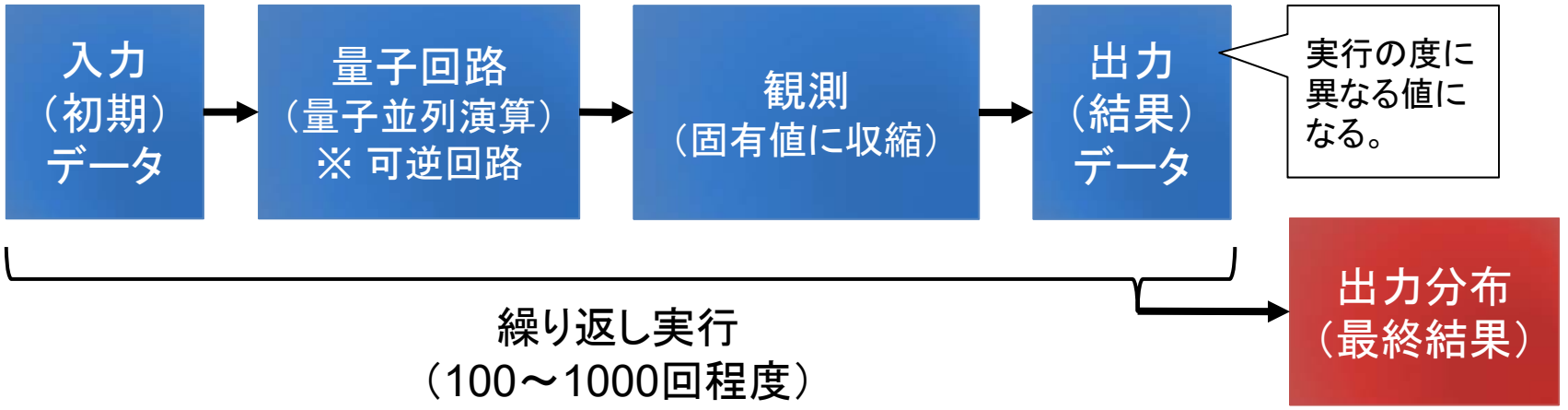
Yだけ見るとXOR

可逆

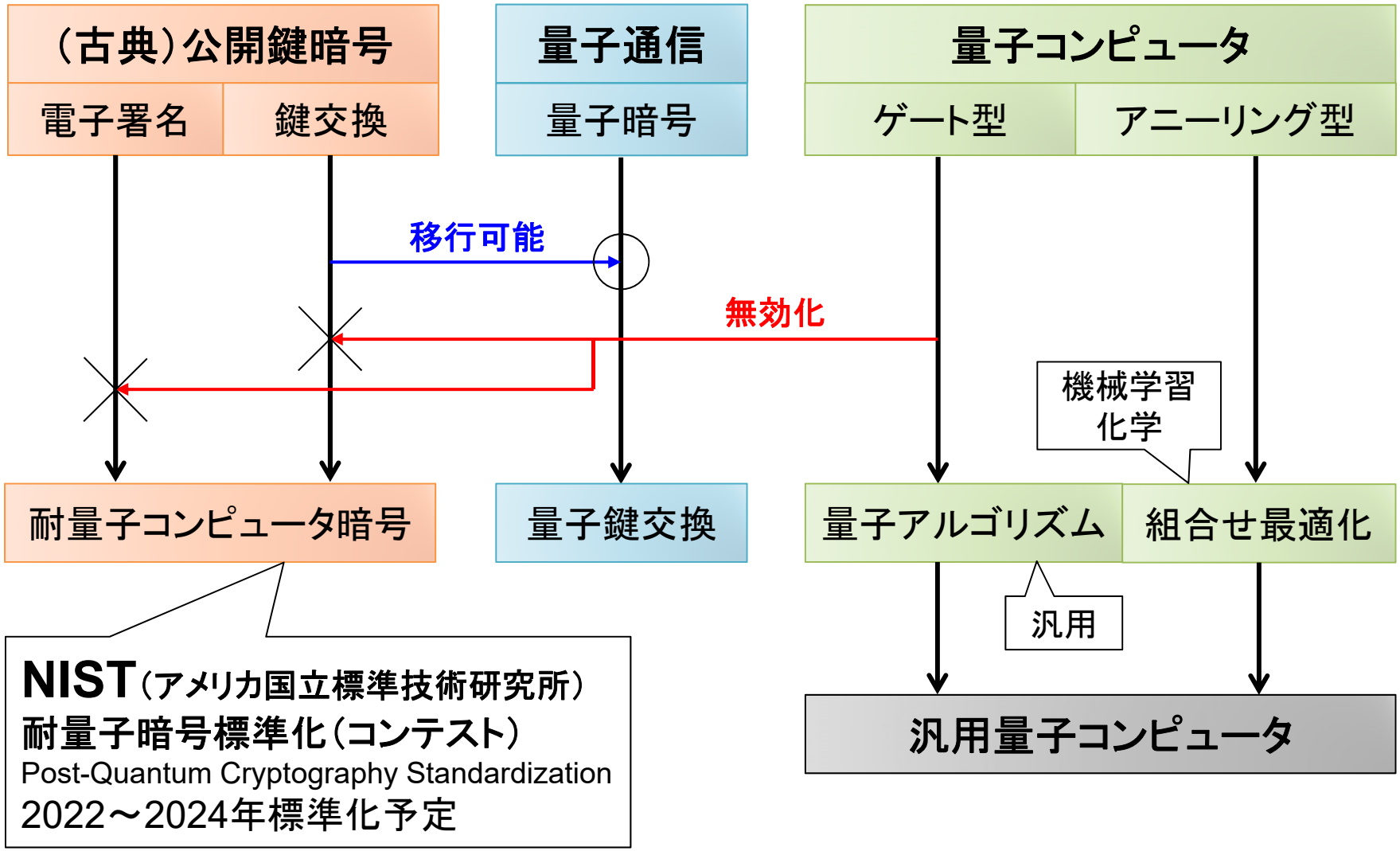
量子コンピュータの概要

1. 0と1の2つの基底を持つ複数の量子ビットを利用。
2. 重ね合わせた状態のままで量子並列演算を行う。
3. 量子ビット同士を絡み合わせて量子回路を構築。
4. 重ね合わせ状態の出力を観測して固有値に収縮。
5. **繰り返し実行**し確率的な固有値の**出力分布**を得る。

→ 二重スリット実験時に光子の分布により干渉縞を確認するようなもの。



暗号と量子の世界


(古典)公開鍵暗号
電子署名
鍵交換
量子通信
量子暗号
量子コンピュータ
ゲート型
アニーリング型
移行可能
無効化
機械学習
化学
耐量子コンピュータ暗号
量子鍵交換
量子アルゴリズム
組合せ最適化
汎用
汎用量子コンピュータ
NIST(アメリカ国立標準技術研究所)
耐量子暗号標準化(コンテスト)
Post-Quantum Cryptography Standardization
2022～2024年標準化予定

量子コンピュータの種類

量子ゲート型（狭義の量子コンピュータ）

方式：量子ゲートの量子回路による量子計算

対象問題：汎用（ただし量子アルゴリズムの範囲内）

開発企業：IBM/Google/Intel/Alibaba/Microsoft等

量子アニーリング型（正確には量子シミュレータ）

方式：イジングモデルを使った量子シミュレーション

対象問題：最適化問題特化（深層学習等への応用）

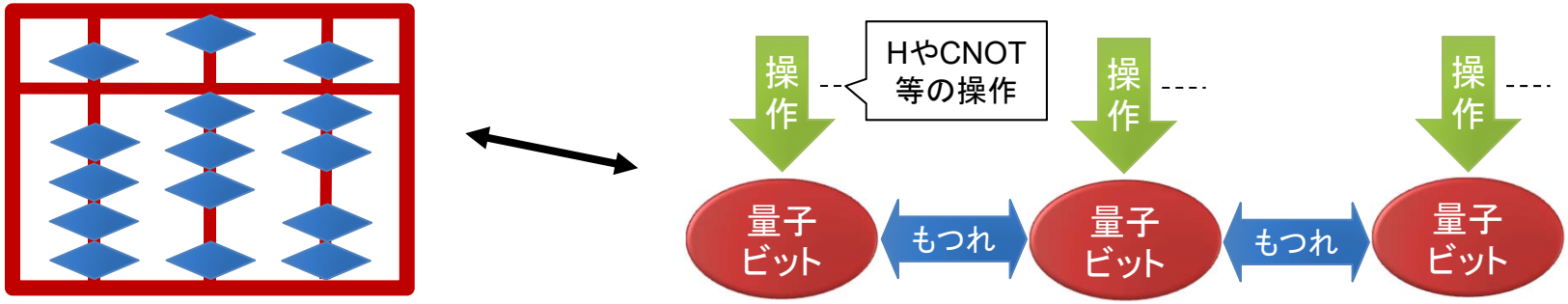
開発企業：D-Wave（非量子型では富士通と日立）

※ 非量子：富士通「デジタルアニーラ」、日立「CMOSアニーリングマシン」。

※ 他に光を使ったCIM（コヒーレントイジングマシン）もあるがここでは省略。

量子ゲート型とソロバン

ソロバンは珠(たま)を配置したハードウェアを、指で弾いて行くことで計算を進めて行く。各珠の間には桁上がり等の関連性がある。



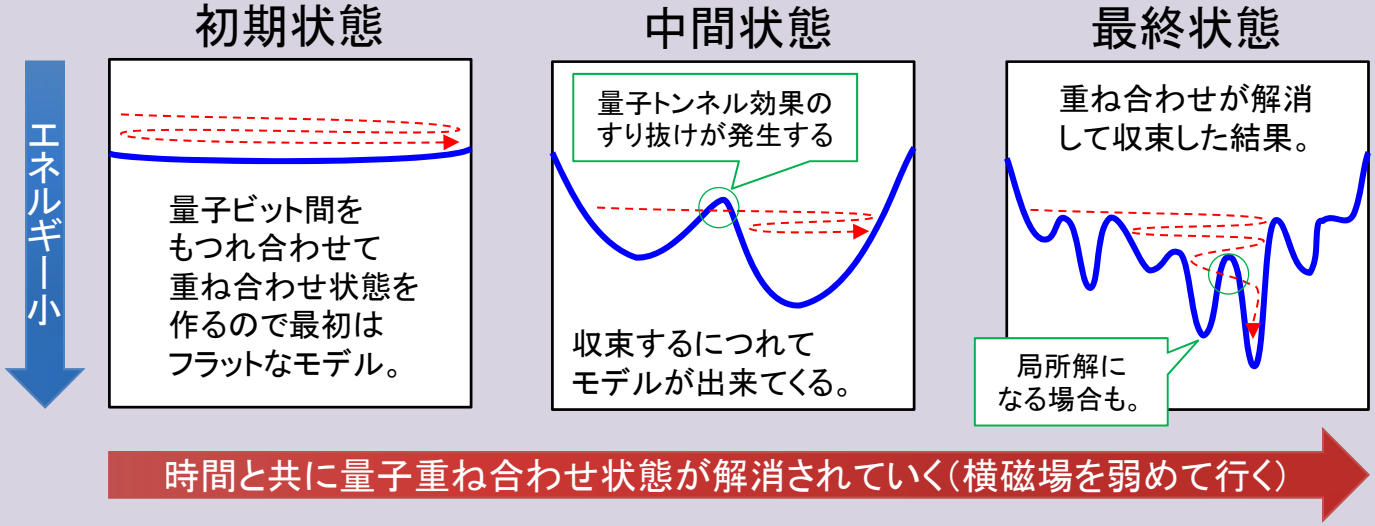
量子ゲート型は量子ビットを配置したハードウェアを、レーザーや電界で弾いて行くことで計算を進めて行く。各量子ビット間には量子もつれによる関連性がある。

ソロバンは可逆回路でもある。またソロバンは同じ操作をすれば毎回同じ値になるが量子では異なる。

量子アニーリング型の計算

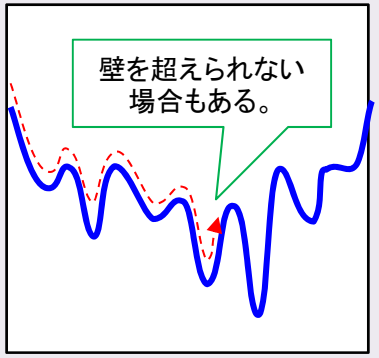
どちらも問題をイジング
モデルとして定式化する
必要があり数学が必要。

量子アニーリング



量子アニーリングでは量子ビット間の全結合(もつれ)が理想だが…

シミュレーテッドアニーリング

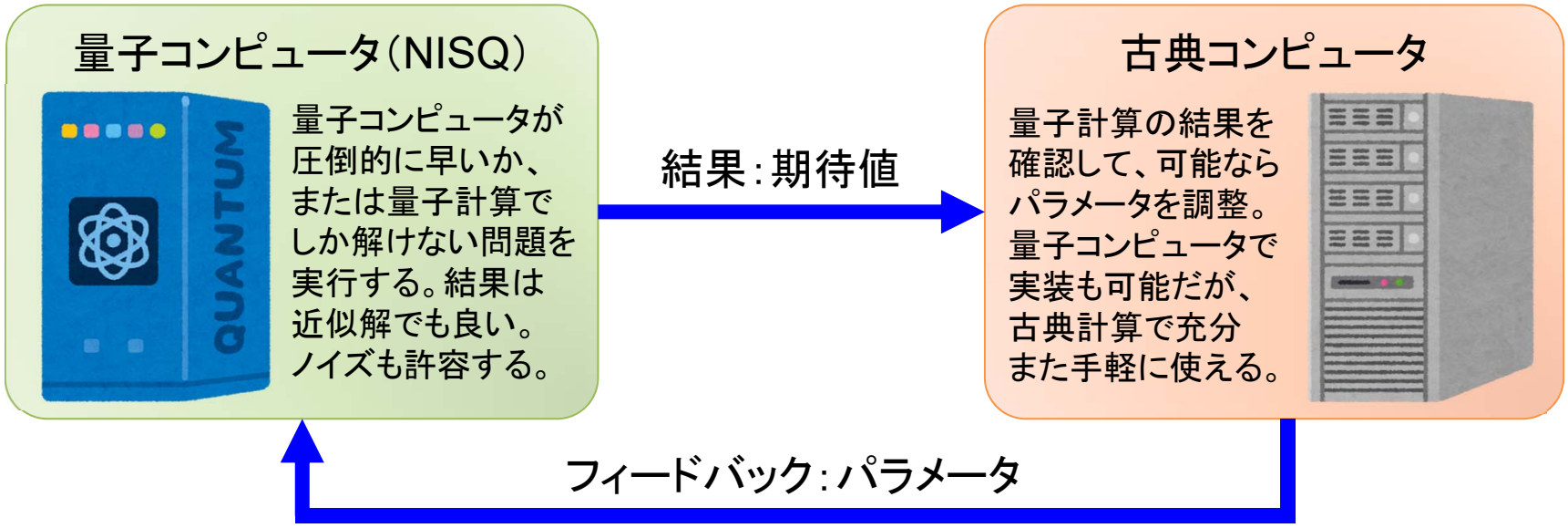


シミュレーテッドアニーリングは先にモデルをセットしてから最適解を求めて行く。

シミュレーテッドアニーリングは古くから使われており、現在でも富士通や日立が専用ハードウェアを使ったアニーリングを実現(例:デジタルアニーラ)している。

古典量子ハイブリッドアルゴリズム

現実的な利用方法として古典コンピュータとの組合せ利用が進んでいる。
後ほど説明するショアのアルゴリズムもある意味ハイブリッド計算である。



主な用途:

- 近似最適化:QAOA(Quantum Approximate Optimization Algo)
- 基底状態探索:VQE(Variational Quantum Eigensolver)
- 機械学習:QCL(Quantum Circuit Learning)

量子コンピュータの未来予想

量子超越
（古典コンピュータでは解けない問題）

RSA暗号解読

どこかでブレークスルーを生じる可能性はゼロではないが…

2015　2018　2023～2030　2040～2060くらい（予測困難）

	NISQ *ノイズあり中規模*	***量子加速*** *（Quantum Speedup）*	***万能（Universal）量子コンピュータ***
実用性 汎用性	ほぼない （一部を除き古典機の方が早い）	一部あり （古典機よりも早い分野がある）	あり （ただし得意不得意はある）
信頼性	ノイズ（エラー）あり	（部分的な）エラー訂正	エラー訂正あり
量子数	$10 \sim 10^2$	（エラー訂正あり）$10^2 \sim 10^6$	10^6（数百万）以上
利用形態	専用機	専用機・外部接続装置	単独・古典機と融合

参考：（ノイマン型）古典コンピュータの歴史

1950　1955　1970

第1世代	***第2世代***	***第3世代***
真空管・試用や研究	*トランジスタ・商用化*	*集積回路・パーソナル化*

量子計算フレームワーク（量子ゲート型）

IBMやGoogleは自社量子コンピュータを使う為の量子計算フレームワークを公開している。実機だけではなくシミュレーション機能を持っているので、量子プログラミングの勉強用として最適だが小規模の量子回路のみとなる。

項目	IBM Qiskit	Google Cirq
ロゴ	 Quantum Information Science Kit	 Cirq
構成	Terra: 量子計算の基盤部（Python） Aqua: 量子アルゴリズムのライブラリ OpenQASM: 量子低レベル中間言語	Cirq:量子計算基盤Pythonライブラリ OpenFermion: 量子化学ライブラリ 量子アニーリング計算も可能
提供	オープンソース（GitHub）	オープンソース（GitHub）
取得	https://qiskit.org/ https://github.com/Qiskit	https://github.com/quantumlib/Cirq
情報	https://qiskit.org/documentation/ja/	https://cirq.readthedocs.io/en/latest/
その他	IBM Q Experience: GUI利用 ※ GUIからOpenQASMに展開し実行 https://quantumexperience.ng.bluemix.net/	2018年夏に正式公開されたライブラリ 量子コンピュータ実機はまだ使えない

量子計算フレームワーク（日本開発）

日本人や日本のベンチャーが開発した量子計算フレームワークも公開されている。まだ歴史は浅いが、後発の分使いやすさを重視しているように見える。

項目	Blueqat	OpenJij
ロゴ	Blueqat	OpenJij
構成	量子ゲート型の計算 量子アニーリング計算も可能	量子アニーリング型の計算
提供	オープンソース（GitHub）	オープンソース（GitHub）
取得	https://github.com/Blueqat	https://github.com/OpenJij
開発	株式会社 MDR https://mdrft.com/?hl=ja	株式会社 Jij https://j-ij.com/
その他	Qiskit/Cirqよりも回路記述が簡単。 前身は量子アニーリング用のWildqat。 日本語のSlackも参加可能。 勉強会・ブログ等で積極的に日本語で情報発信をしている。	2019年に本格的に公開開始した。 定式化された数式で入力可能。 日本語のチュートリアルも完備。 日本語のSlackも参加可能。 現在Windows上では動作しない。

Part 1：関連数学と1量子ビット操作

（振り返り短縮版）

関連数学は省略するので必要な方は
前回（Part1）の元資料を見てください。

波動力学と行列力学

時間発展しないシュレディンガー方程式（波動力学）

$$H\varphi(x) = E\varphi(x)$$

Hはハミルトニアン式、φ(x)は波動関数（固有関数）、Eはエネルギー固有値

行列における固有値問題の式

$$Ax = \lambda x$$

Aは正方行列、xは列ベクトル（固有ベクトル）、λはスカラー値（固有値）

量子計算は行列の固有値問題として解くことができる。
これをハイゼンベルク方程式（行列力学）と呼ばれる。
この場合に波動関数は固有ベクトルとして求められる。

※ 波動力学と行列力学が数学的に同じであることはシュレディンガーにより確認されている。

物理学と数学の違い

紛らわしい…特にA^*（スター）…
このページ以降は物理学の
作法で記述して行きます。

	物理学 （量子力学）	数学 （線形代数学）
複素共役 （虚数部の±反転）	A^* （スター）	$\overline{A}$ （バー）
複素共役転置	$A^\dagger$ （ダガー）	A^* （スター）
エルミート （自己随伴）	$A = A^\dagger$	$A = A^*$
ユニタリ	$U^{-1} = U^\dagger$	$U^{-1} = U^*$
単位行列	I （id:単位ゲート）	E

ブラケット（BraKet）記法

⟨Bra| ：ブラ（行）ベクトル
|Ket⟩ ：ケット（列）ベクトル

n次元ブラ：（行）ベクトル $\langle\varphi| = \begin{pmatrix} d_0 & d_1 & \cdots & d_n \end{pmatrix}$ ，

n次元ケット：（列）ベクトル $|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$

量子計算では
基底ベクトルとして
1 と 0 の2次元の
ブラケットを利用。

$\langle 0| = \begin{pmatrix} 1 & 0 \end{pmatrix}$

$\langle 1| = \begin{pmatrix} 0 & 1 \end{pmatrix}$

$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

量子計算結果の
|0⟩ と |1⟩ が重要

ブラケットベクトルの内積と外積

n次φブラ：（行ベクトル） $\langle\varphi| = \begin{pmatrix} d_0 & d_1 & \cdots & d_n \end{pmatrix}$ ，n次ψケット：（列ベクトル） $|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$

がある時に、

内積：$\langle\varphi|\psi\rangle = \begin{pmatrix} d_0 & d_1 & \cdots & d_n \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} = d_0c_0 + d_1c_1 + \cdots + d_nc_n = \sum_{j=0}^{n} d_jc_j$

結果はスカラー

外積：$|\psi\rangle\langle\varphi| = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} \begin{pmatrix} d_0 & d_1 & \cdots & d_n \end{pmatrix} = \begin{pmatrix} c_0d_0 & c_0d_1 & \cdots & c_0d_n \\ c_1d_0 & c_1d_1 & \cdots & c_1d_n \\ \vdots & \vdots & \ddots & \vdots \\ c_nd_0 & c_nd_1 & \cdots & c_nd_n \end{pmatrix}$

結果は行列

テンソル積（1階のテンソル積）

2つの2次元ベクトル $|\psi_1\rangle$ と $|\psi_2\rangle$ がある時、

$$|\psi_1\rangle = \begin{pmatrix} a \\ b \end{pmatrix} = a|0\rangle + b|1\rangle \qquad |\psi_2\rangle = \begin{pmatrix} c \\ d \end{pmatrix} = c|0\rangle + d|1\rangle$$

$|\psi_1\rangle$ と $|\psi_2\rangle$ のテンソル積⊗は4次元ベクトルとなる。

$$|\psi_1\rangle \otimes |\psi_2\rangle = \begin{pmatrix} a\begin{pmatrix} c \\ d \end{pmatrix} \\ b\begin{pmatrix} c \\ d \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix} = \begin{array}{l} ac|00\rangle + ad|01\rangle \\ \quad + bc|10\rangle + bd|11\rangle \end{array}$$

テンソル積の記号⊗は省略されることが多い。

$$|\psi_1\rangle \otimes |\psi_2\rangle \quad \Rightarrow \quad |\psi_1\rangle|\psi_2\rangle \quad \Rightarrow \quad |\psi_1\psi_2\rangle$$

複数量子ビットとテンソル積

複数量子ビットは
テンソル積で表せる。

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$ であるので、

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$|000\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad |001\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{...(略)...} \quad |111\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$|111\rangle = |7\rangle$
と表示する場合もある

有限準位量子力学系（量子ビット＝2次元）

量子ビットは2つの固有ベクトル･固有値を持つ。
2つの固有（基底）ベクトルを $|0\rangle$ と $|1\rangle$ とする。

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

任意の量子ベクトル ψ に関して確率振幅 c_n を
使ってボルンの規則により以下の式が成り立つ、

確率振幅
c_0 と c_1 は
複素数

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle$$

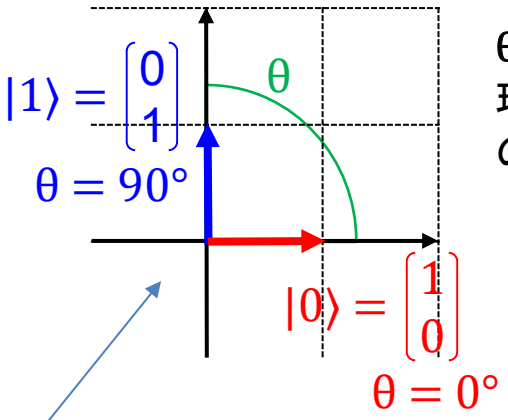
$$|c_0|^2 + |c_1|^2 = 1$$

$$c_0 = \langle 0|\psi\rangle \qquad c_1 = \langle 1|\psi\rangle$$

重要!

量子ビットとブロッホ球

基底ベクトル



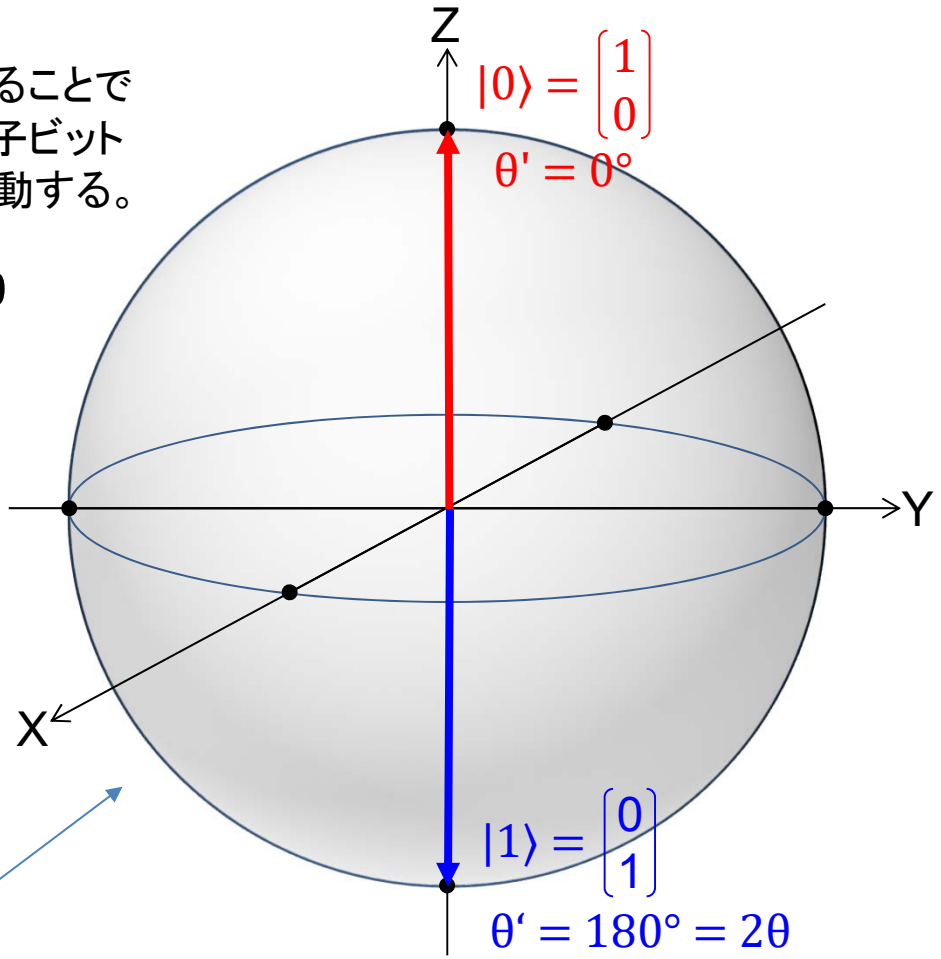
これでは上半球の部分しか
使えない…また虚数部がある
ので4次元の空間となる。
量子状態のイメージが掴めない…

θ‘ = 2θ とすることで
球面上を量子ビット
の状態が移動する。

$\theta' = 2\theta$

ブロッホ球

量子状態を単位球面上に表す表記法



虚数部（位相）は絶対値ではなく
干渉に関する相対値が必要。
なので$|0\rangle$の位置をゼロとして、
$|1\rangle$の虚数部（Z軸の回転θ）に
位相差を示し3次元空間にする。

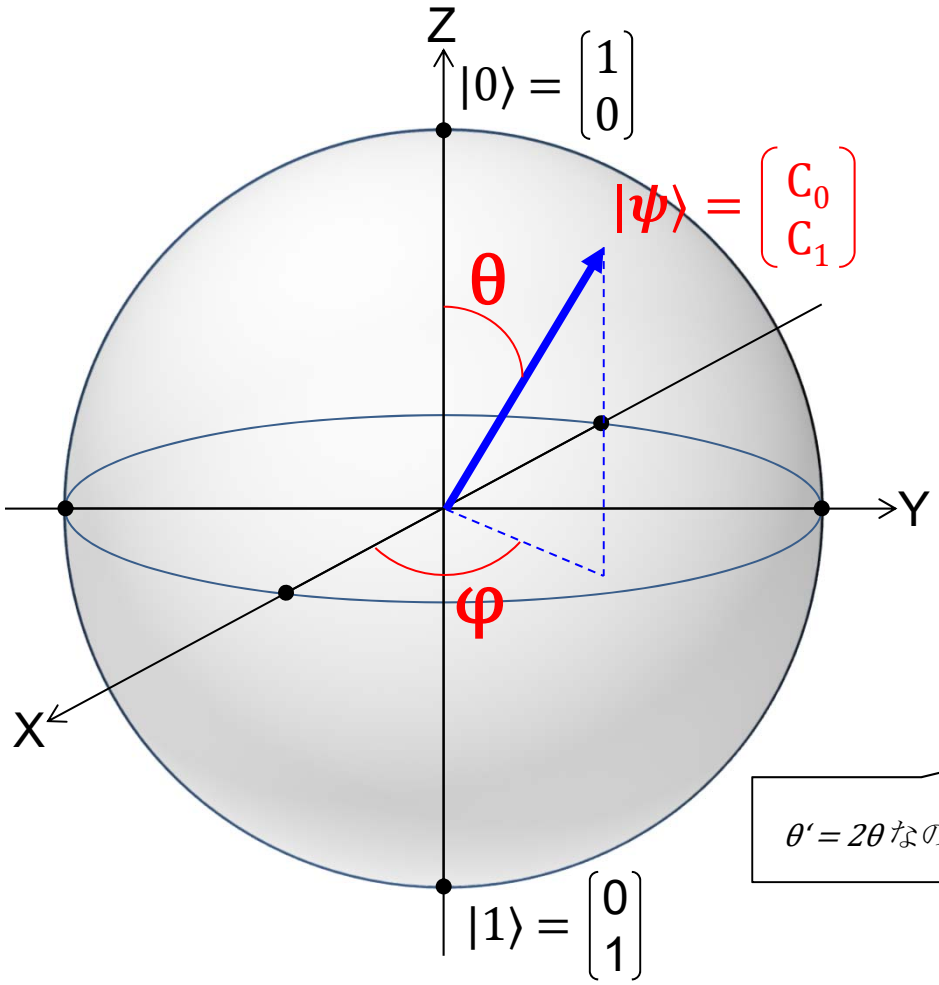
ブロッホ球は複素ヒルベルト空間を示す。

量子ビットの存在確率と位相

重要!

ブロッホ球

量子状態を単位球面上に表す表記法



$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle$$

$$|c_0|^2 + |c_1|^2 = 1$$ 球面上の制約

$\|0\rangle$ 実数部	$z = \cos\theta$
$\|0\rangle$ 虚数部	位相差を見るのでゼロ
$\|1\rangle$ 実数部	$x = \cos\varphi \sin\theta$
$\|1\rangle$ 虚数部	$y = \sin\varphi \sin\theta$

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + (\cos\varphi + i\sin\varphi)\sin\frac{\theta}{2}|1\rangle$$

位相差

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle$$

$\theta' = 2\theta$ なので $\frac{\theta}{2}$

二乗すると $|0\rangle$ の存在確立

二乗すると $|1\rangle$ の存在確立

※ θ は存在確率を、φ は位相を示す。

量子ビットに対するユニタリ演算

ユニタリ行列による座標変換: $U|\psi\rangle = |\psi'\rangle$

id 恒等演算 $\mathrm{iden}(q) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

単位行列 I なので何もしない演算となる。

Cirqでは未サポート

X ビット反転演算 $\mathrm{X}(q) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

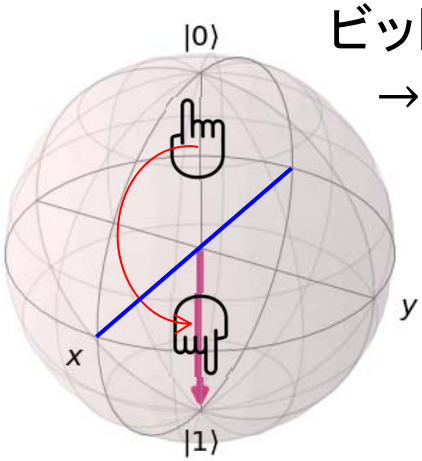
Y 位相ビット反転演算 $\mathrm{Y}(q) = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$

Z 位相反転演算 $\mathrm{Z}(q) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

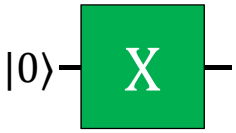
パウリ (Pauli) ゲート

位相のみを変換する場合は位相シフトゲートとも呼ぶ。

パウリ X/Y/Z ゲート（各種反転操作）

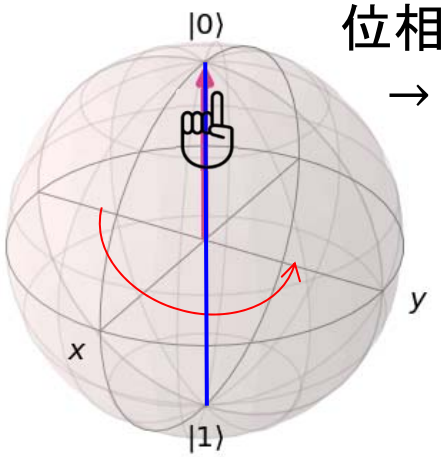


ビット反転演算 x(q)
→ X軸回転

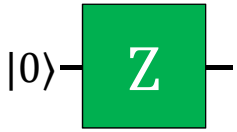


cirq.X(Q)

X |0⟩ = |1⟩
X |1⟩ = |0⟩

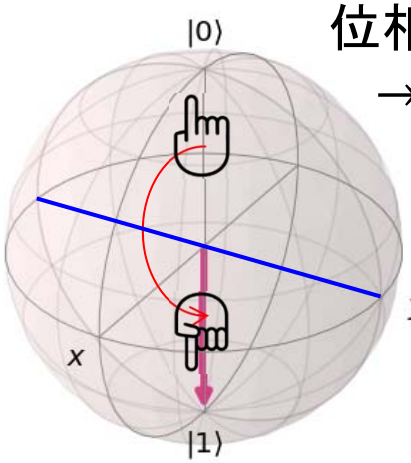


位相反転演算 z(q)
→ Z軸回転

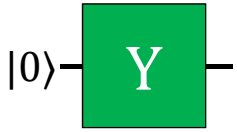


cirq.Z(Q)

X |0⟩ = |0⟩
X |1⟩ = −|1⟩



位相ビット反転演算 y(q)
→ Y軸回転



cirq.Y(Q)

X |0⟩ = i|1⟩
X |1⟩ = −i|0⟩

アダマール H ゲート（重ね合わせ状態の生成）

アダマールによる座標変換: $H|\psi\rangle = |\psi'\rangle$

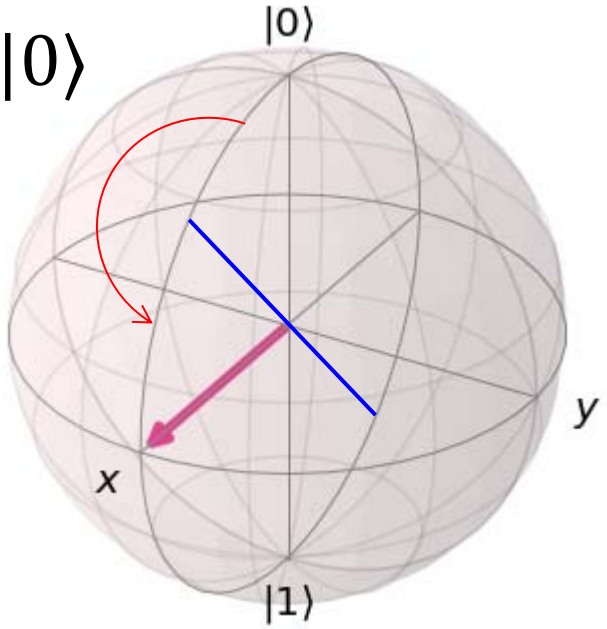
H　アダマール演算 $H(q) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

Hadamard

$= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$

cirq.H(Q)

$H|0\rangle$

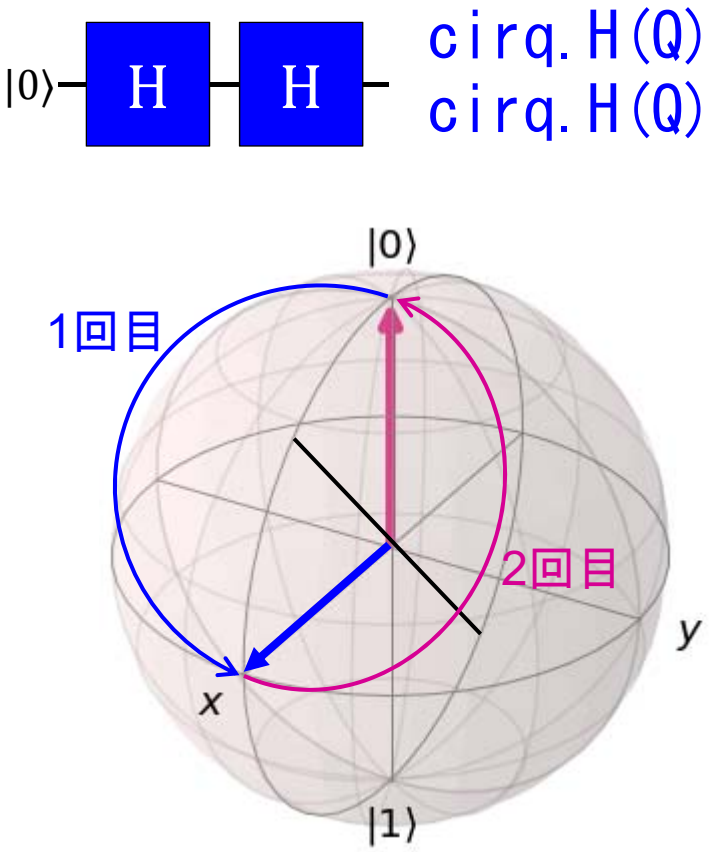


重要!

Hゲートの演算は、
ブロッホ球のXZ平面に、
45度傾いた軸回りに、
180度（π）回転する。

※ Y軸に90度回転する訳ではない。

アダマール H ゲートを2回適用



$$HH = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I \text{（単位行列）}$$

$$HH\,|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

結論：2回アダマール演算を適用すると元のベクトルに戻る。

量子ゲートの組み合わせ

- アダマールゲートで挟むと変換が可能となる。

$HXH = Z$ ：ビット反転Xゲートを挟むと位相反転Zゲートに。

$HZH = X$ ：位相反転Zゲートを挟むとビット反転Xゲートに。

$HYH = -Y$ ：位相ビット反転Yゲートを挟むと位相のみ反転。

ex) HXHの計算

$$XH = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$H(XH) = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = Z$$

位相シフト S/S†/T/T† ゲート

Z　位相反転演算 $Z(q) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

反転なので逆位相も同じ

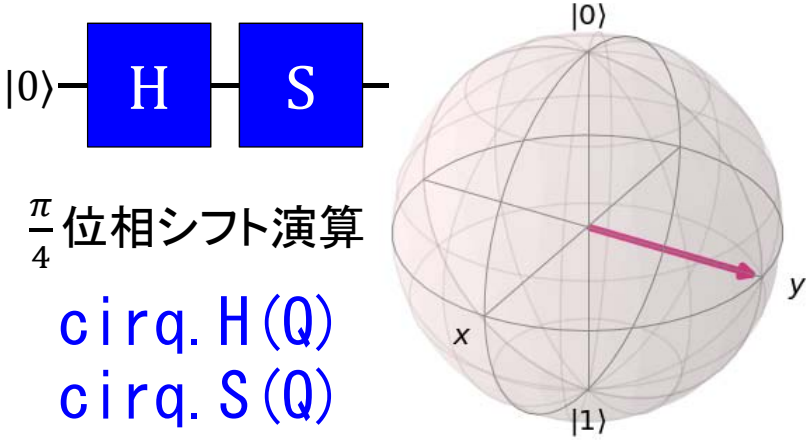
S　$\frac{\pi}{4}$ 位相シフト演算 $S(q) = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$

S†　$-\frac{\pi}{4}$ 位相シフト演算 $S^{\dagger}(q) = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$

T　$\frac{\pi}{8}$ 位相シフト演算 $T(q) = \begin{pmatrix} 1 & 0 \\ 0 & \frac{(1+i)}{\sqrt{2}} \end{pmatrix}$

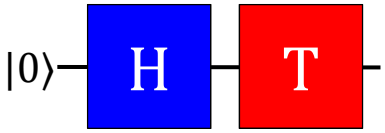
T†　$-\frac{\pi}{8}$ 位相シフト演算 $T^{\dagger}(q) = \begin{pmatrix} 1 & 0 \\ 0 & \frac{(1-i)}{\sqrt{2}} \end{pmatrix}$

重ね合わせ状態での位相シフト



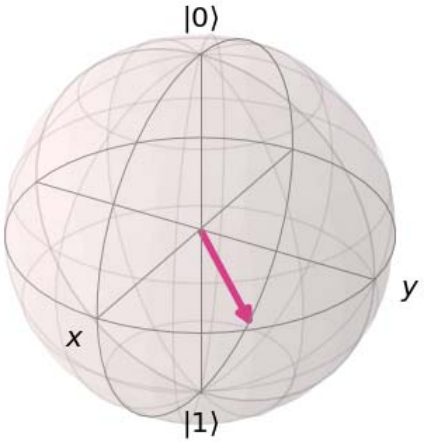
$\frac{\pi}{4}$位相シフト演算

```
cirq.H(Q)
cirq.S(Q)
```

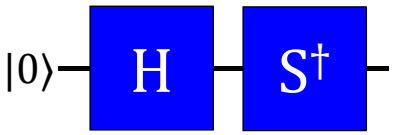



$\frac{\pi}{8}$位相シフト演算

```
cirq.H(Q)
cirq.T(Q)
```

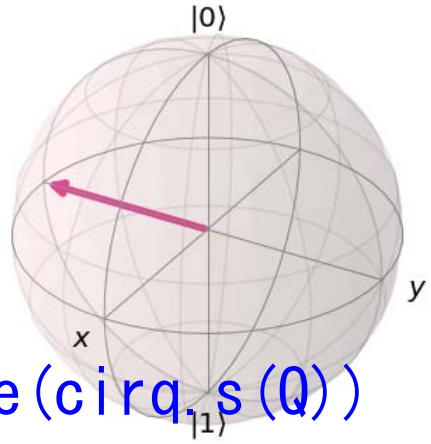


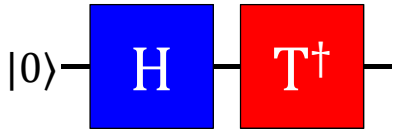
※ ダガー"†"が付くと逆方向の位相となる。



$-\frac{\pi}{4}$位相シフト演算

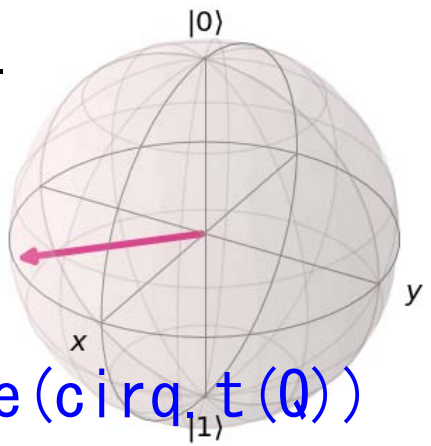
```
cirq.H(Q)
cirq.inverse(cirq.s(Q))
```





$-\frac{\pi}{8}$位相シフト演算

```
cirq.H(Q)
cirq.inverse(cirq.t(Q))
```



軸回転 Rx/Ry/Rz ゲート

実際の量子プログラミングでは、任意角度での軸回転（シフト）ゲートが必要になる。

Rx X軸回転演算 $Rx(\theta, q) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$

Ry Y軸回転演算 $Ry(\theta, q) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$

Rz Z軸回転演算 $Rz(\theta, q) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$

測定（量子ビットを収束させて名前を付けて保存）

—M('m')—	標準基底測定 measure(q, key='m')

Cirq におけるゲート表示（テキスト表示）

—X—	Xゲート：cirq.X(q)	—S—	Sゲート：cirq.S(q)
—Y—	Yゲート：cirq.Y(q)	—T—	Tゲート：cirq.T(q)
—Z—	Zゲート：cirq.Z(q)	—Rx(f)—	Rxゲート：cirq.Rx(f).on(q)
—H—	Hゲート：cirq.H(q)	—Ry(f)—	Ryゲート：cirq.Ry(f).on(q)
—@—	制御ゲート（後述）	—Rz(f)—	Rzゲート：cirq.Rz(f).on(q)

GoogleのCirqを使う

環境：Anaconda3（Python3.5）

以下より環境に合わせてダウンロードとインストール

https://www.anaconda.com/distribution/

ライブラリ：Cirq（シルク）

Windows版：Anaconda Prompt

MacOS版：ターミナル

インストール

```
pip install cirq
```

バージョン指定インストール

```
pip install cirq=0.5.0
```

アンインストール

```
pip uninstall cirq
```

※ Cirqのバージョン確認：

In:	import cirq cirq.__version__
Out:	'0.5.0‘

本資料のソースは 0.5.0 と表示される環境にて確認しています。

Cirq アダマール演算（量子 Hello World!）

Jupiter Notebook から以下を実行(コピペで大丈夫です)。

```
import cirq                                    # cirq量子計算用
Q = cirq.LineQubit(0)                          # 量子ビットを1つ準備
qc = cirq.Circuit.from_ops(                    # 量子回路生成
  cirq.H(Q),                                   # 量子重ね合わせ（アダマール演算）
  cirq.measure(Q, key='m')                     # 量子ビットQを観測しkey名'm'へ.
)
print("Circuit:")
print(qc)                                      # 量子回路表示
qsim = cirq.Simulator()                        # 量子シミュレータ
result = qsim.run(qc, repetitions=1000)        # 回路を1000回実行する
print("Results:")
result.histogram(key='m')                      # key名'm'のヒストグラム表示
```

実行結果。

```
Circuit:
(0, 0): ———H———M('m')———
Results:
Counter({0: 494, 1: 506})
```

プログラマの為の量子コンピュータ入門 目次:

Part 1: 関連数学と1量子ビット操作

- 線形代数学の基本知識
- ブラケット記法と量子計算
- ブロッホ球と1量子ビット操作（IBM Qiskit）

Part 2: 量子ゲート型のプログラミング

- 2量子ビット以上の量子回路（量子もつれ）
- 量子アルゴリズム（グローバ検索・フーリエ変換等）
- 量子ゲート型プログラミング（Google Cirq）

今回範囲

Part 3: 量子アニーリング型のプログラミング

- イジングモデルとQUBOと量子アニーリング
- 量子アルゴリズム（セールスマン巡回問題）
- 量子アニーリング型プログラミング（D-Wave Ocean）

Part 2：量子ゲート型のプログラミング

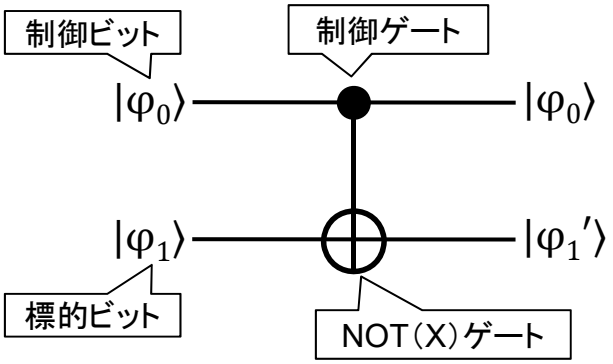
2-1：複数量子ビット操作

前回（Part1）は1量子ビット操作の話をしました。
（ここまでは大丈夫ですか？）

今回はまず量子ゲート型の複数量子ビット操作を説明して行きます。

制御反転 CNOT（CX）ゲート

重要!



CNOTゲートの演算は、制御（Controlled）ビットが $|1\rangle$ の時のみ標的（Targeted）ビットが反転する。

制御反転演算 CNOT(q0, q1) $=$ $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

CXとしても良い

$\|\varphi_0\rangle$	$\|\varphi_1\rangle$	$\|\varphi_0\rangle$	$\|\varphi_1'\rangle$
$\|0\rangle$	$\|0\rangle$	$\|0\rangle$	$\|0\rangle$
$\|0\rangle$	$\|1\rangle$	$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$	$\|1\rangle$	$\|1\rangle$
$\|1\rangle$	$\|1\rangle$	$\|1\rangle$	$\|0\rangle$

テンソル積

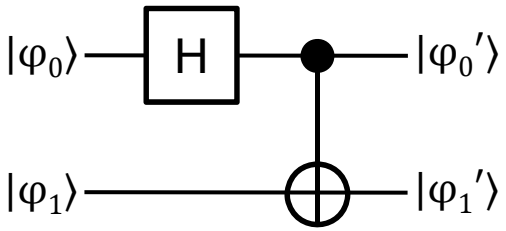
CNOT $|00\rangle = |00\rangle$
CNOT $|01\rangle = |01\rangle$
CNOT $|10\rangle = |11\rangle$
CNOT $|11\rangle = |10\rangle$
反転

出力 $|\varphi_1'\rangle$ だけ見ると XOR ゲートと言える

ベル状態（量子もつれ）

重要!

ベル状態とは、2つの量子ビット間に量子もつれを生じている状態であり、1つを観測するともう片方の値が決まる。
量子回路としてはアダマールとCNOTを使って実現可能。



ベル状態（量子もつれ）回路

$$|\varphi_0\varphi_1\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\end{pmatrix} \\ \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{pmatrix}$$

$$|\varphi_0'\varphi_1'\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

入力：$|\varphi_0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, $|\varphi_1\rangle = |0\rangle$

出力：$|\varphi_0'\varphi_1'\rangle = \frac{1}{\sqrt{2}}|00\rangle + \boxed{0|01\rangle + 0|10\rangle} + \frac{1}{\sqrt{2}}|11\rangle$

$= 0\%$

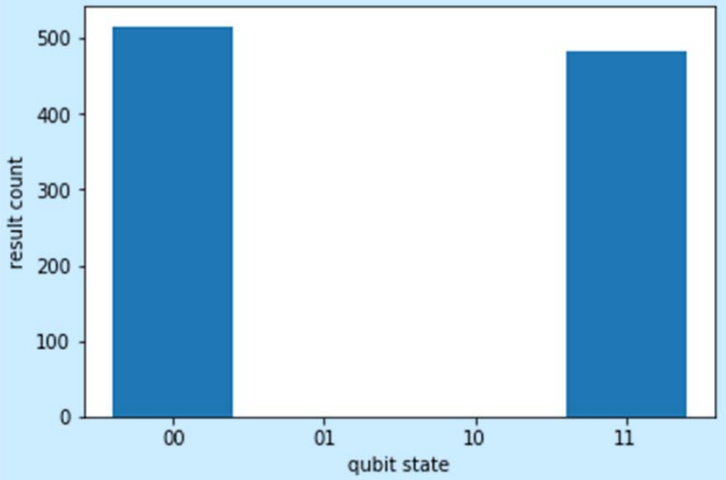
※ $|\varphi_0\rangle$が$|0\rangle$で観測されると$|\varphi_1'\rangle$も$|0\rangle$に、$|\varphi_0\rangle$が$|1\rangle$で観測されると$|\varphi_1'\rangle$も$|1\rangle$となる。

ベル状態回路の実行

```
import cirq
Q = [cirq.LineQubit(i) for i in range(2)]
qc = cirq.Circuit.from_ops(
  cirq.H(Q[0]),
  cirq.CNOT(Q[0],Q[1]),
  cirq.measure(Q[0], key='q0'),
  cirq.measure(Q[1], key='q1')
)
print("Circuit:")
print(qc)
qsim = cirq.Simulator()
result = qsim.run(qc, repetitions=1000)
print("Result:")
cirq.plot_state_histogram(result)
```

```
Circuit:
0: ──────H──────@──────M('q0')──────
                │
1: ─────────────X──────M('q1')──────
```

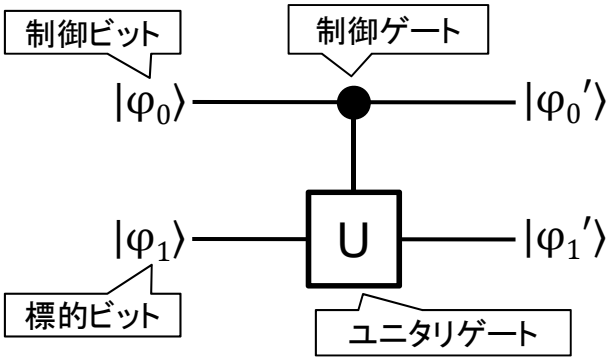
実行結果



array([516., 0., 0., 484.])

q0が0ならq1も0に、
q0が1ならq1も1になる。
01と10にはならない。
ベル状態（量子もつれ）
を確認できる。

制御ユニタリ CU ゲート



CX(CNOT)ゲートは標的ビット側をXゲートにしたが、YやZ,H,S等の任意ユニタリ演算も指定できる。

$$CU = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{pmatrix}$$

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = Y \qquad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = Z$$

ここに任意のユニタリ演算行列を入れる

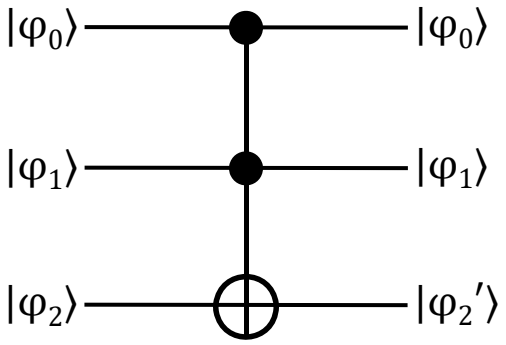
例）Zゲートの CZ(q0, q1) ベル演算結果：$\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$

```
cirq.H(Q0) / cirq.CZ(Q0,Q1)
Circuit:
0: ———H———@———
          |
1: ———————@———
```

CZゲートは上下を入れ替えても同じ。回路ではどちらも“@”で表現される。

$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

トフォリ CCNOT ゲート



CCNOT (Toffoli)

```
cirq.CCNOT(Q0, Q1, Q2)
Circuit:
0: ──────@──────
         |
1: ──────@──────
         |
2: ──────X──────
```

制御反転ゲートに対しもう1つ制御ビットを追加したCCNOTゲート。

トフォリ演算 CCNOT (q0, q1, q2) =

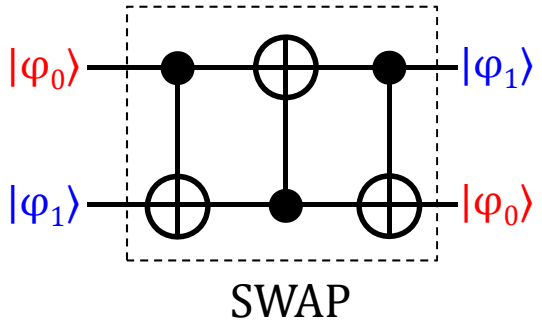
$$\begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \end{pmatrix}$$

CSWAP $|000\rangle = |000\rangle$　CSWAP $|100\rangle = |100\rangle$
CSWAP $|001\rangle = |001\rangle$　CSWAP $|101\rangle = |101\rangle$
CSWAP $|010\rangle = |010\rangle$　CSWAP $|110\rangle = |111\rangle$
CSWAP $|011\rangle = |011\rangle$　CSWAP $|111\rangle = |110\rangle$

反転

※トフォリゲートはAND/OR/NOT等が実現可能な万能ゲートである。

交換 SWAP ゲート



=

```
cirq.SWAP(Q0, Q1)
Circuit:
0: ———×———
      |
1: ———×———
```

Xでは無い

CNOTゲートを3つ
交互に重ねると
量子ビットの値を
交換できる。

交換演算 $SWAP(q0, q1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

$SWAP\ |00\rangle = |00\rangle$
$SWAP\ |01\rangle = |10\rangle$
$SWAP\ |10\rangle = |01\rangle$
} 反転
$SWAP\ |11\rangle = |11\rangle$

計算過程:

$|\varphi_0\rangle\ |\varphi_1\rangle \rightarrow$

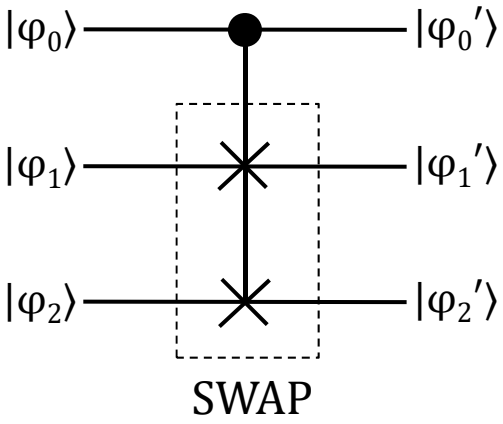
⊕ は XOR

$(CNOT_{01}) \rightarrow |\varphi_0\rangle\ |\varphi_1 \oplus \varphi_0\rangle$

$(CNOT_{10}) \rightarrow |\varphi_0 \oplus (\varphi_1 \oplus \varphi_0)\rangle\ |\varphi_1 \oplus \varphi_0\rangle = |\varphi_1\rangle\ |\varphi_1 \oplus \varphi_0\rangle \rightarrow$

$(CNOT_{01}) \rightarrow |\varphi_1\rangle\ |(\varphi_1 \oplus \varphi_0) \oplus \varphi_1\rangle = |\varphi_1\rangle\ |\varphi_0\rangle$

制御交換 CSWAP ゲート




```
cirq.CSWAP(Q0, Q1, Q2)
Circuit:
0: ———@———
      |
1: ———×———
      |
2: ———×———
```

交換ゲートに対し
制御ビットを追加
したCSWAPゲート。

制御交換演算 CSWAP(q0, q1, q2) =

$$\begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1 \end{pmatrix}$$

CSWAP $|000\rangle = |000\rangle$　CSWAP $|100\rangle = |100\rangle$
CSWAP $|001\rangle = |001\rangle$　CSWAP $|101\rangle = |110\rangle$
CSWAP $|010\rangle = |010\rangle$　CSWAP $|110\rangle = |101\rangle$
CSWAP $|011\rangle = |011\rangle$　CSWAP $|111\rangle = |111\rangle$

反転

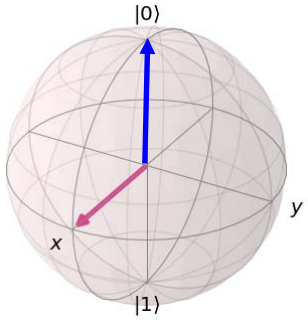
2-2: 量子アルゴリズムの基本

複数の量子ビットを使った演算も行えるようになりました。これをどう使えば量子計算ができるのかを疑問に思われているでしょう。

まず量子アルゴリズムの基本となる、位相に関係する仕組みを勉強します。

プラスケット $|+\rangle$ とマイナスケット $|-\rangle$

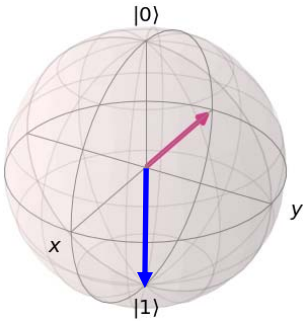
$|+\rangle$ は、$|0\rangle$ をアダマール変換した重ね合わせ状態。
$|+\rangle$ を再度アダマール変換すると $|0\rangle$ に戻る。



$$|+\rangle = \mathrm{H}|0\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$$

$$\mathrm{H}|+\rangle = |0\rangle$$

$|-\rangle$ は、$|1\rangle$ をアダマール変換した重ね合わせ状態。
$|-\rangle$ を再度アダマール変換すると $|1\rangle$ に戻る。
$|+\rangle$ と $|-\rangle$ は逆位相の関係。



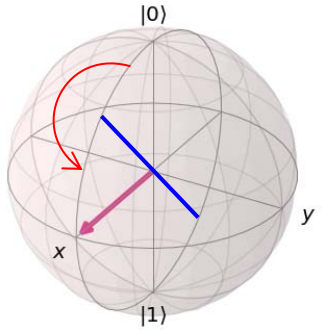
$$|-\rangle = \mathrm{H}|1\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix} = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$$

$$\mathrm{H}|-\rangle = |1\rangle$$

アダマール変換と位相

プラスケット $|+\rangle$ と マイナスケット $|-\rangle$ も考慮すると、
アダマール変換では、入力ケット（下の例では $|\varphi\rangle$）が、
位相に影響すると言う見方もできる。

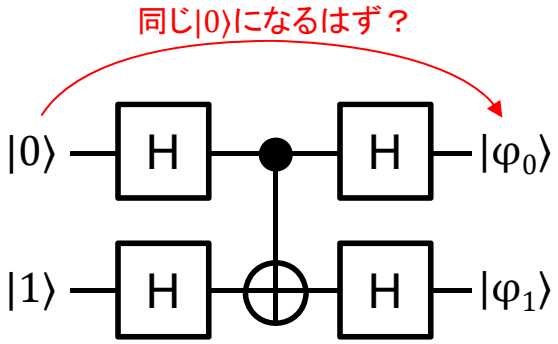
$$H|\varphi\rangle = \frac{1}{\sqrt{2}}|0\rangle + (-1)^{\varphi}\frac{1}{\sqrt{2}}|1\rangle$$



重要!

アダマール変換は、
重ね合わせ状態の生成と同時に、
振幅（入力ベクトル）を位相に変換する。

制御反転 CNOTゲートと位相の反転



CNOTゲートの制御(Controlled)ビットは
同じ値が出力される…はず？
ではアダマールで重ね合わせした値
を入力するとどうなる？

答え：実行すると $|\varphi_0\rangle = |1\rangle$, $|\varphi_1\rangle = |1\rangle$ になる。
あれ？なぜ制御ビットの値が変化しているの？

$$|0\rangle \otimes |1\rangle \rightarrow (\mathrm{H}\otimes\mathrm{H}) \rightarrow |+\rangle \otimes |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes|-\rangle$$

$$\rightarrow (\mathrm{CNOT}) \rightarrow \frac{1}{\sqrt{2}}((-1)^{\mathrm{NOT}(0)}|0\rangle+(-1)^{\mathrm{NOT}(1)}|1\rangle)\otimes|-\rangle$$

$$= \frac{1}{\sqrt{2}}(-|0\rangle+|1\rangle)\otimes|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)\otimes|-\rangle = |-\rangle\otimes|-\rangle$$

$$\rightarrow (\mathrm{H}\otimes\mathrm{H}) \rightarrow |1\rangle \otimes |1\rangle$$

制御ビットの値は変化しない
が位相が反転する。

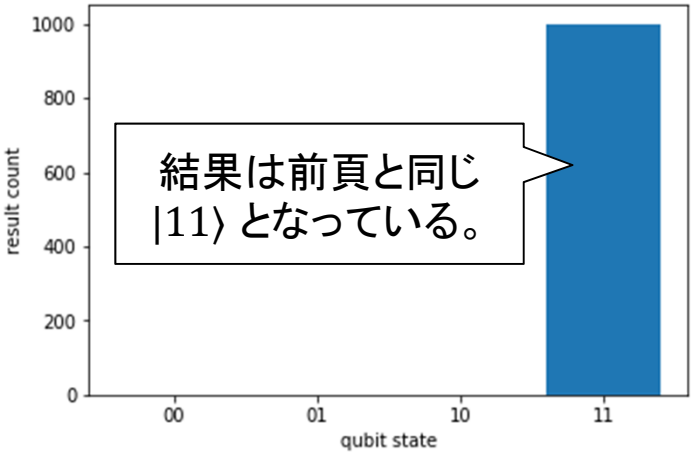
アダマールとCNOTの確認

Jupiter Notebook から以下回路を実行。

```
Q = [cirq.LineQubit(i) for i in range(2)]  # 2量子ビットを用意
qc = cirq.Circuit.from_ops(
  cirq.X(Q[1]),                          # Q1を反転して |1⟩にする
  cirq.H.on_each(*Q),                    # Q0/Q1にアダマール変換
  cirq.CNOT(Q[0],Q[1]),                  # CNOT適用
  cirq.H.on_each(*Q),                    # Q0/Q1にアダマール変換
  cirq.measure(Q[0], key='q0'),          # Q0を観測
  cirq.measure(Q[1], key='q1')           # Q1を観測
)
```

実行結果。

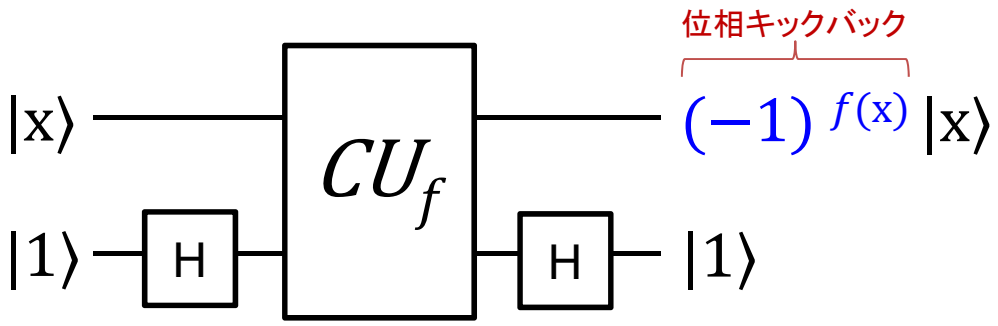
```
Circuit:
0: ———————H———@———H———M('q0')———
                |
1: ———X———H———X———H———M('q1')———
Results:
array([   0.,    0.,    0., 1000.])
```



位相キックバック（Phase kick back）

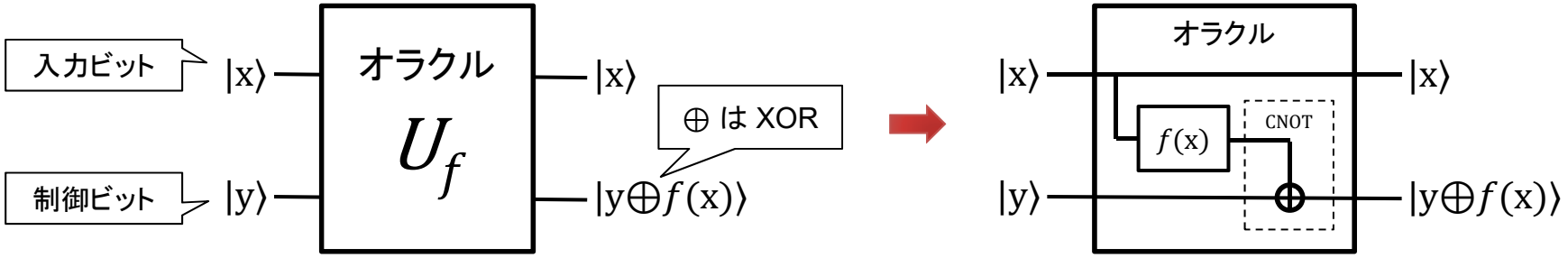
前頁のCNOT（CX）では位相が制御ビットに影響したが
これは一般的な制御ユニタリCUゲートにも適用可能。

CUゲートの動作を関数 $f(x)$ とすると、
関数を制御ビットの位相に反映する。
これを**位相キックバック**と呼ぶ。



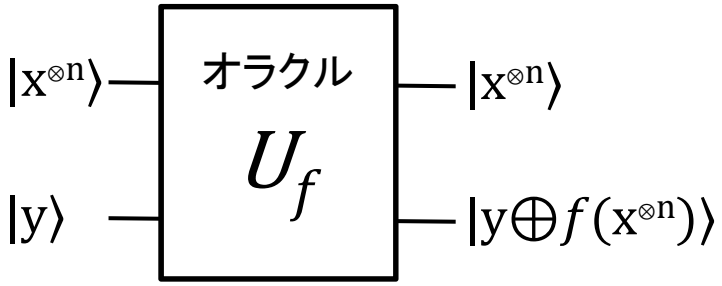
オラクル(Oracle:神託)ボックス

関数 $f(x)$ を計算するオラクル(神託)ボックス U_f を想定。



オラクルボックス U_f はユニタリ演算のブラックボックス。

入力は複数ビットに拡張できる。



2-3: ドイチェ アルゴリズム

位相キックバックとオラクルボックスが分かったところで、オラクルボックスを使う基本として、ドイチェアルゴリズムを見ます。

ドイチェ(Deutsch)問題

ドイチェ問題は**1ビット関数 $f(x)$ の性質(種類)を判定**。

性質A:一定(constant):出力は常に一定の値となる。
$f(0)=0$ かつ $f(1)=0$ 、または $f(0)=1$ かつ $f(1)=1$

性質B:均等(balanced):0 と 1 が均等(50%)の確率で出力される。
$f(0)=0$ かつ $f(1)=1$ 、または $f(0)=1$ かつ $f(1)=0$

	性質A:一定(constant)		性質B:均等(balanced)	
$f(0)$	0	1	0	1
$f(1)$	0	1	1	0

古典アルゴリズムでは $f(0)$ と $f(1)$ の両方結果を取得しないと、どちらの性質(種類)か判定ができない。古典アルゴリズムでは $f(0)=0$ だったとしても $f(1)$ の結果を見る必要がある。

つまり**問い合わせが $f(0)$ と $f(1)$ の必ず2回必要**となる。

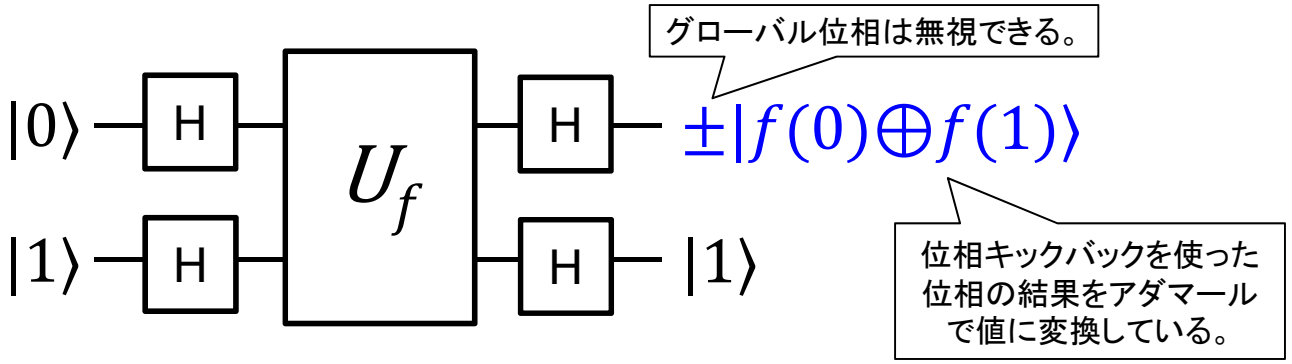
ドイチェ問題の量子計算

入力を $|0\rangle$ と $|1\rangle$ とする。

重ね合わせ状態を作る。

測定ユニタリの入出力にアダマールHを適用。

上位ビット出力は $f(0)$ と $f(1)$ の $\oplus$(XOR)となる。

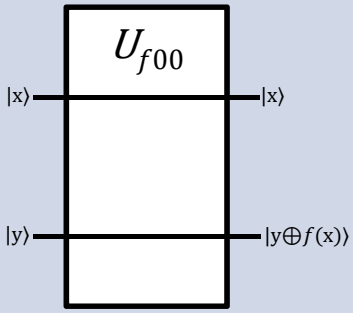
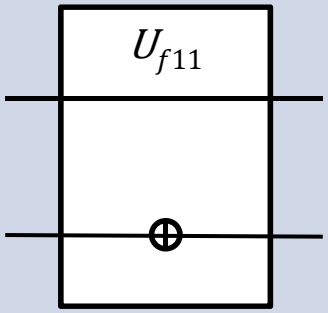
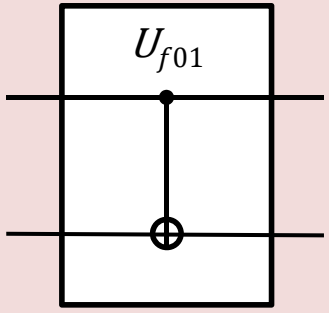
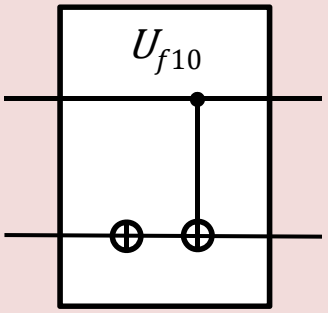


上位ビット出力 $f(0) \oplus f(1)$ を確認して判定。

一定(constant) : $f(0) \oplus f(1) = |0\rangle$

均等(balanced) : $f(0) \oplus f(1) = |1\rangle$

ドイチェ問題を計算する為の定義

	性質A：一定（constant）		性質B：均等（balanced）	
	$f_{00}(x)$	$f_{11}(x)$	$f_{01}(x)$	$f_{10}(x)$
$f(0)$	**0**	**1**	**0**	**1**
$f(1)$	**0**	**1**	**1**	**0**
f 変換	$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
U_f ゲート				
cirq	def oracle_f00(x, y_fx) return yield 何もしない	def oracle_f11(x, y_fx) yield X(y_fx)	def oracle_f01(x, y_fx) yield CNOT(x, y_fx)	def oracle_f10(x, y_fx) yield X(y_fx) yield CNOT(x, y_fx)

ドイチェ問題の計算

Jupiter Notebook から以下回路を実行(左右は連続)。

```
from cirq import *
# ユニタリ定義
def oracle_f00(x, y_fx):
    return
    yield
def oracle_f01(x, y_fx):
    yield CNOT(x, y_fx)
def oracle_f10(x, y_fx):
    yield X(y_fx)
    yield CNOT(x, y_fx)
def oracle_f11(x, y_fx):
    yield X(y_fx)
```

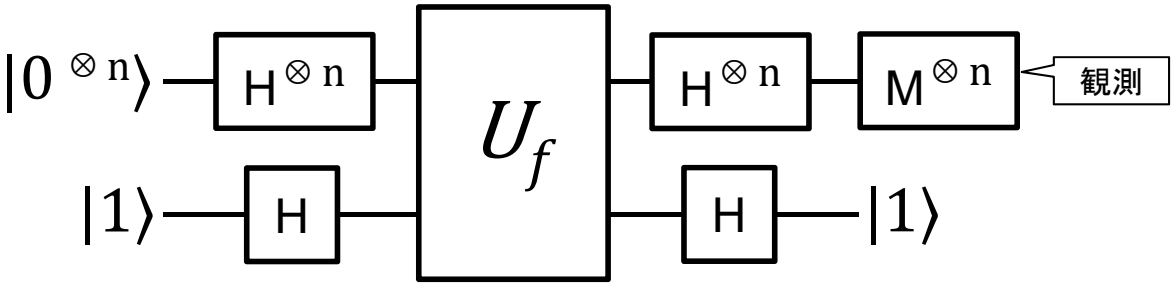
```
Q = [LineQubit(i) for i in range(2)]
qc = Circuit()
qc.append(X(Q[1]))
qc.append(H.on_each(*Q))
qc.append(oracle_f00(Q[0], Q[1]))
# qc.append(oracle_f01(Q[0], Q[1]))
# qc.append(oracle_f10(Q[0], Q[1]))
# qc.append(oracle_f11(Q[0], Q[1]))
qc.append(H.on_each(*Q))
qc.append(measure(Q[0], key='c'))
r = Simulator().run(qc)
print(r)
```

実行結果。

```
qc.append(oracle_f00(Q[0], Q[1]))  ⇒ (実行結果) ⇒  c=0  (constant)
qc.append(oracle_f01(Q[0], Q[1]))  ⇒ (実行結果) ⇒  c=1  (balanced)
qc.append(oracle_f10(Q[0], Q[1]))  ⇒ (実行結果) ⇒  c=1  (balanced)
qc.append(oracle_f11(Q[0], Q[1]))  ⇒ (実行結果) ⇒  c=0  (constant)
```

ドイチェ・ジョサ(Deutsch-Jozsa)問題

ドイチェ問題の入力を複数量子ビットにした問題。
2ビット以上だと一定・均等以外のケースもある。
例:2ビットの時に 0010 のように一定でも均等でも無いケースがあり得る。
この為に一定・均等いずれかであることを前提。



複数ビットに対しても適用可能とすることで、
ビット数が増えても演算は1回で済む。

2ビットのドイチェ・ジョサ問題

	一定（constant）		均等（balanced）					
	f_{C0}	f_{C1}	f_{B0}	f_{B1}	f_{B2}	f_{B3}	f_{B4}	f_{B5}
$f(00)$	0	1	0	0	0	1	1	1
$f(01)$	0	1	0	1	1	1	0	0
$f(10)$	0	1	1	0	1	0	1	0
$f(11)$	0	1	1	1	0	0	0	1

$f_{C0}(x1, x2) = 0$ ，　$f_{C1}(x1, x2) = 1$

$f_{B0}(x1, x2) = x1$ ，　$f_{B1}(x1, x2) = x2$

$f_{B2}(x1, x2) = x1 \oplus x2$ ，　$f_{B3}(x1, x2) = \mathrm{NOT}(x1)$

$f_{B4}(x1, x2) = \mathrm{NOT}(x2)$ ，　$f_{B5}(x1, x2) = \mathrm{NOT}(x1 \oplus x2)$

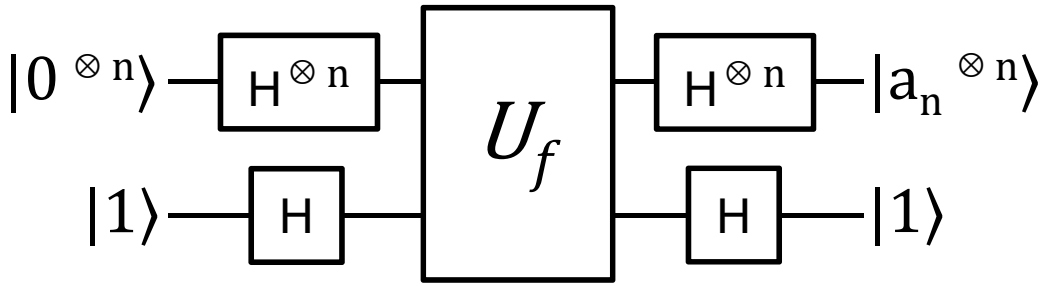
2ビットまではCNOT/NOTゲートで構成可能。
3ビット以上ではCCNOT（トフォリ）ゲートが必要。

その他の量子アルゴリズム問題

➢ ベルンシュタイン・ヴァジラニ(Bernstein-Vazirani)問題

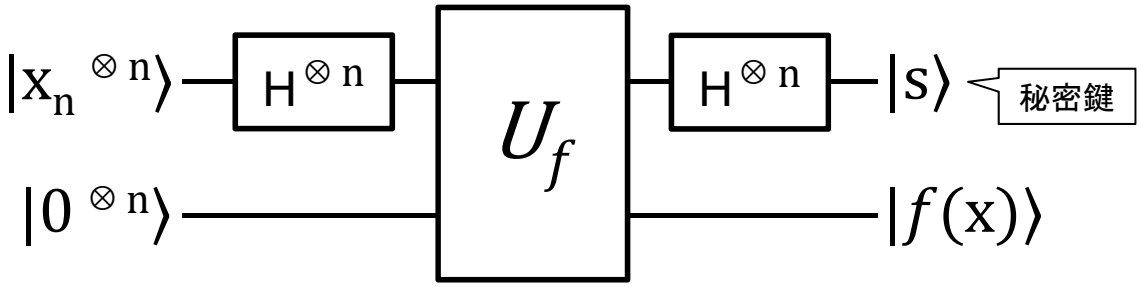
nビットの変数xと定数aの内積を解とする関数$f(x)$がある時に定数aを求める。

$f(x)=x\cdot a=(x_0\cdot a_0)\oplus(x_1\cdot a_1)\oplus\ldots\oplus(x_n\cdot a_n)$



➢ サイモン(Simon)問題

nビット変数xと関数$f(x)$がある時に、$f(x)=f(x\oplus s)$となる秘密鍵sを得る問題。
ただしs≠0とする。古典では2^n回かかるが量子ではn回で済む。



ドイチェ系 量子アルゴリズム

1. アダマールゲートによる重ね合わせを作る。
   - 複数の入力状態を同時に実現する。
   - 振幅(値)を位相に変換する。
2. ブラックボックスにて位相キックバックを使う。
   - 内容が分からないユニタリ変換関数の性質を得る問題用。
   - 関数を使ったオラクルボックスを設定する。
   - 関数の内容を位相にキックバックさせる。
3. アダマールゲートにより位相を振幅に戻す。
   - 位相を振幅(値)に変換する。

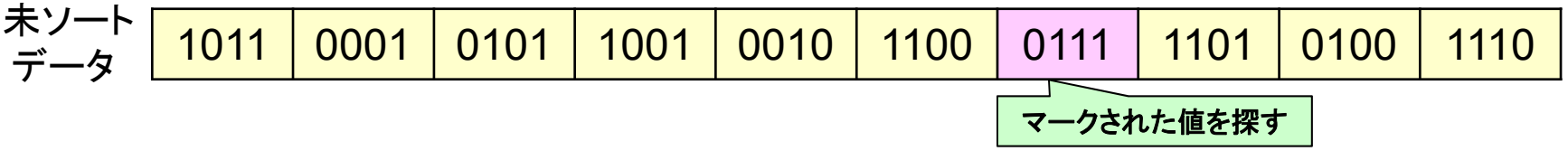
※ 量子アルゴリズムを使うことで、対象となるビット数が増えても1回の問合せで済む。これらの問題では確率的な解にはならず決定的な解となる。

2-3: グローバー検索（量子検索）

検索を高速化する重要な量子アルゴリズムが、
グローバー検索です。

グローバー(Grover)検索問題

n個の未ソート(ランダム)状態のデータがある時、
解となる特定の値xを検索する問題である。



古典的な計算ではn回の検索が必要となるが、
グローバー(量子)検索では$\sqrt{n}$回の検索で済む。
※ 量子検索でも1回では検索できないが充分早い。

検索と言っているが、関数 $y=f(x)$ がある時に、
特定の解 y を与える x が存在するかどうかを、
判断する逆関数の導出問題である。

振幅増幅手法

欲しい解の確率振幅をマイナスにマーキングして、全確率振幅の平均値で逆転させる計算手法で、グローバー検索で利用。

問題: $|\varphi\rangle = \frac{1}{2}(|00\rangle+|01\rangle+|10\rangle+|11\rangle)$ がある時に、
この中から $|10\rangle$ を探す（2量子ビットにおける検索例）。

手順1: 解 $|10\rangle$ の確率振幅（位相）をマイナスにマーキングする。

$$|\varphi\rangle_{\text{marked}} = \frac{1}{2}(|00\rangle+|01\rangle-|10\rangle+|11\rangle)$$

手順2: 全確率振幅の平均値を求める。

$$\langle a\rangle = \left(\frac{1}{2}+\frac{1}{2}-\frac{1}{2}+\frac{1}{2}\right)/4 = \frac{1}{4}$$

平均値の周りで逆転する為に、
平均値から確率振幅を引いた後で平均値を足す

手順3: 全確率振幅の平均値の周りで反転させる。

$$|\varphi\rangle'_{\text{marked}} = \left(\frac{1}{4}-\frac{1}{2}+\frac{1}{4}\right)|00\rangle + \left(\frac{1}{4}-\frac{1}{2}+\frac{1}{4}\right)|01\rangle + \left(\frac{1}{4}+\frac{1}{2}+\frac{1}{4}\right)|10\rangle + \left(\frac{1}{4}-\frac{1}{2}+\frac{1}{4}\right)|11\rangle = |10\rangle$$

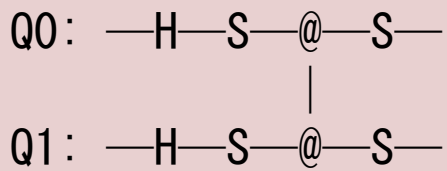
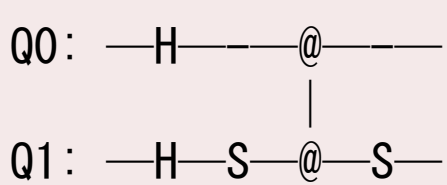
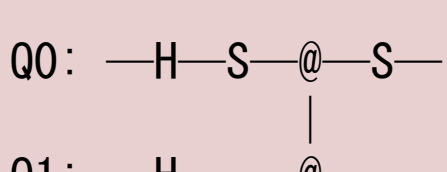
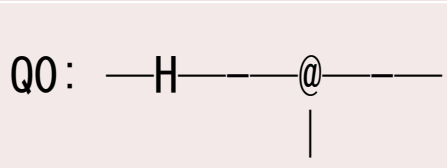
求める $|10\rangle$ が
見つかった

|10⟩ のマーキング実行

```
from cirq import *
import numpy as np          # 結果表示にnumpyを使う
Q = [LineQubit(i) for i in range(2)]
qc = Circuit()
qc.append(H.on_each(*Q))    # 均等な重ね合わせ状態を作る
qc.append(S(Q[0]))
qc.append(CZ(*Q))
qc.append(S(Q[0]))
print("Circuit:")
print(qc)
r = Simulator().simulate(qc)     # 位相を見る
print(np.around(r.final_state, 3))  # 丸めて結果表示
```

```
Circuit:
0: ———H———S———@———S———
               |
1: ———H———————@—————————
[ 0.5+0.j  0.5+0.j -0.5+0.j  0.5-0.j]
```

2量子ビット確率分布(位相)マーキング

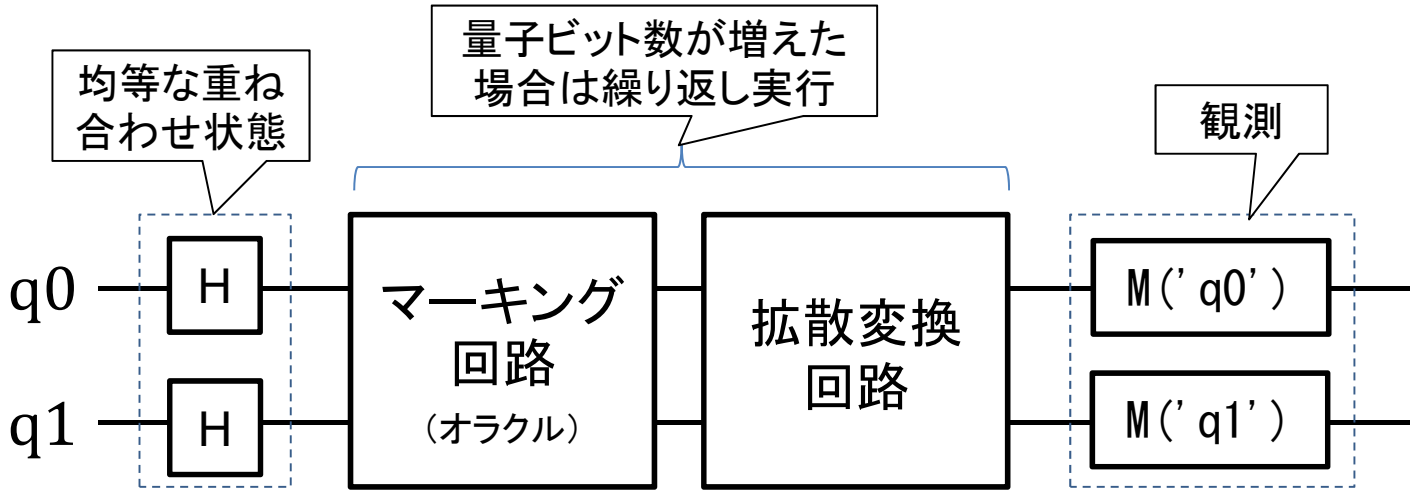
対象	回路	回路ソース	実行結果
|00⟩		H.on_each(*Q) S.on_each(*Q) CZ(*Q) S.on_each(*Q)	[0.5+0.j -0.5+0.j -0.5+0.j -0.5+0.j]
|01⟩		H.on_each(*Q) S(Q[1]) CZ(*Q) S(Q[1])	[0.5+0.j -0.5+0.j 0.5+0.j 0.5-0.j]
|10⟩		H.on_each(*Q) S(Q[0]) CZ(*Q) S(Q[0])	[0.5+0.j 0.5+0.j -0.5+0.j 0.5-0.j]
|11⟩		H.on_each(*Q) CZ(*Q)	[0.5+0.j 0.5+0.j 0.5+0.j -0.5+0.j]

平均値の周りで反転（拡散変換）

```
0: ―H―X―@―X―H―
          |
1: ―H―X―@―X―H―
```

平均値の周りで反転させる（拡散変換）回路

グローバー検索の回路

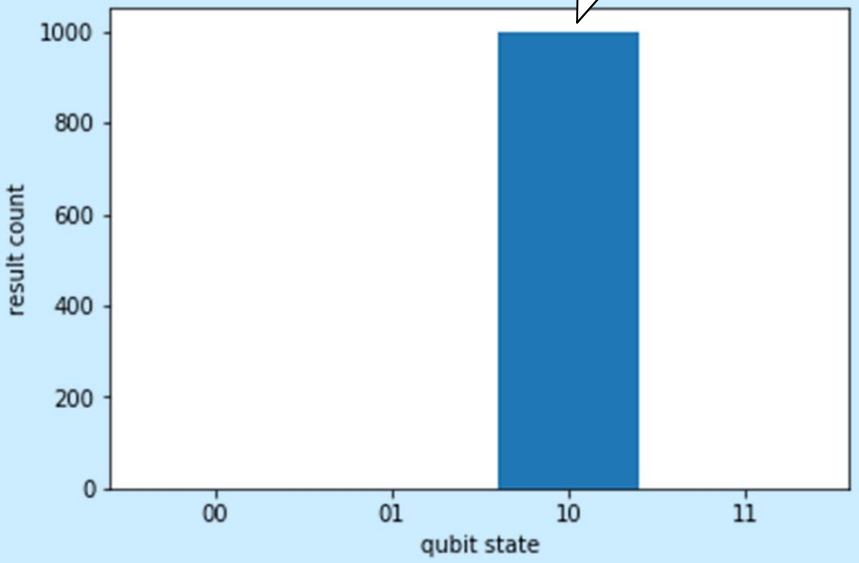


2量子ビットのグローバー検索の実行

```
from cirq import *
Q = [LineQubit(i) for i in range(2)]
qc = Circuit()
# 均等な重ね合わせ状態を作る
qc.append(H.on_each(*Q))
# マーキング |10>
qc.append(S(Q[0]))
qc.append(CZ(*Q))
qc.append(S(Q[0]))
# 平均値の周りで反転 （拡散変換）
qc.append(H.on_each(*Q))
qc.append(X.on_each(*Q))
qc.append(CZ(*Q))
qc.append(X.on_each(*Q))
qc.append(H.on_each(*Q))
# 観測 （結果）
qc.append(measure(Q[0], key='q0'))
qc.append(measure(Q[1], key='q1'))
# 実行と結果表示
r = Simulator().run(qc, repetitions=1000)
plot_state_histogram(r)
```

解として|10⟩ が出力された

実行結果



array([0., 0., 1000., 0.])

繰り返し数の最適回数

今回は2量子ビットの問題を解いたので1回の拡散変換で結果が出たが、量子ビット数が増えると繰り返して拡散変換を実行する必要がある。なお最適回数をオーバーすると結果が悪くなる。

繰り返し数の最適回数Kは以下の式となる。

$$\text{最適回数 K} = \frac{\pi}{4}\sqrt{N} - \frac{1}{2}$$

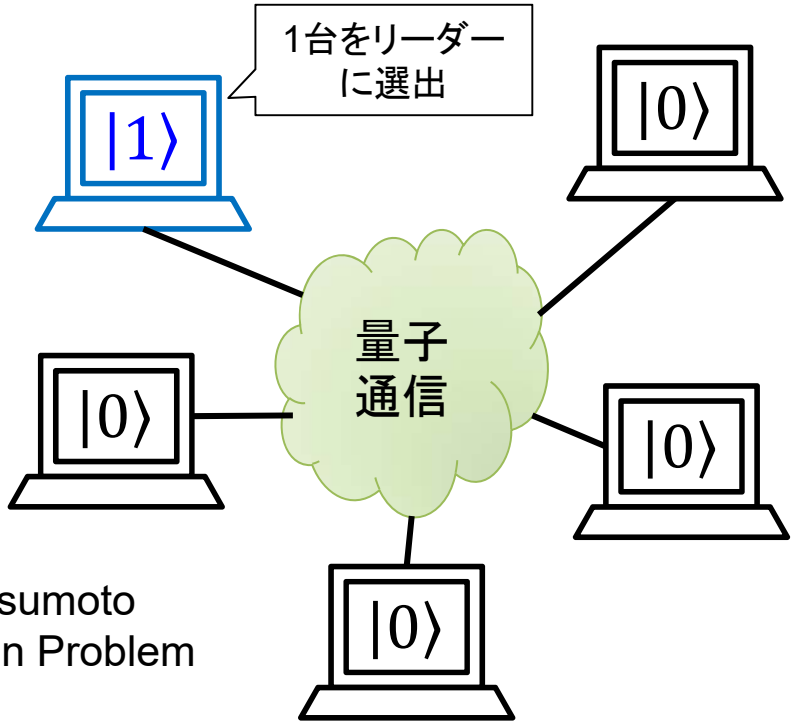
N	計算結果	K
2^2=4	1.070796327…	1
2^3=8	1.721441469…	2
2^4=16	2.641592654…	3
2^5=32	3.942882938…	4
2^6=64	5.783185307…	6

N	計算結果	K
2^8=256	12.06637061…	12
2^{10}=1024	24.63274123…	25
2^{12}=4096	49.76548246…	50
2^{14}=16384	100.0309649…	100
2^{16}=65536	200.5619298…	201

リーダー選挙問題

識別不能な端末がn台存在するネットワーク上でリーダー端末を1台選出する問題。古典では端末の識別子があると簡単だが、識別不能の場合は有限時間内に確率が1にならない問題となる。

本問題は各端末が量子コインを振り1つだけが $|1\rangle$ となる場合を得る問題と解釈できる。
量子通信を使って分散オラクルを実現することで量子探索問題により、有限時間内に確率を1とすることができる。

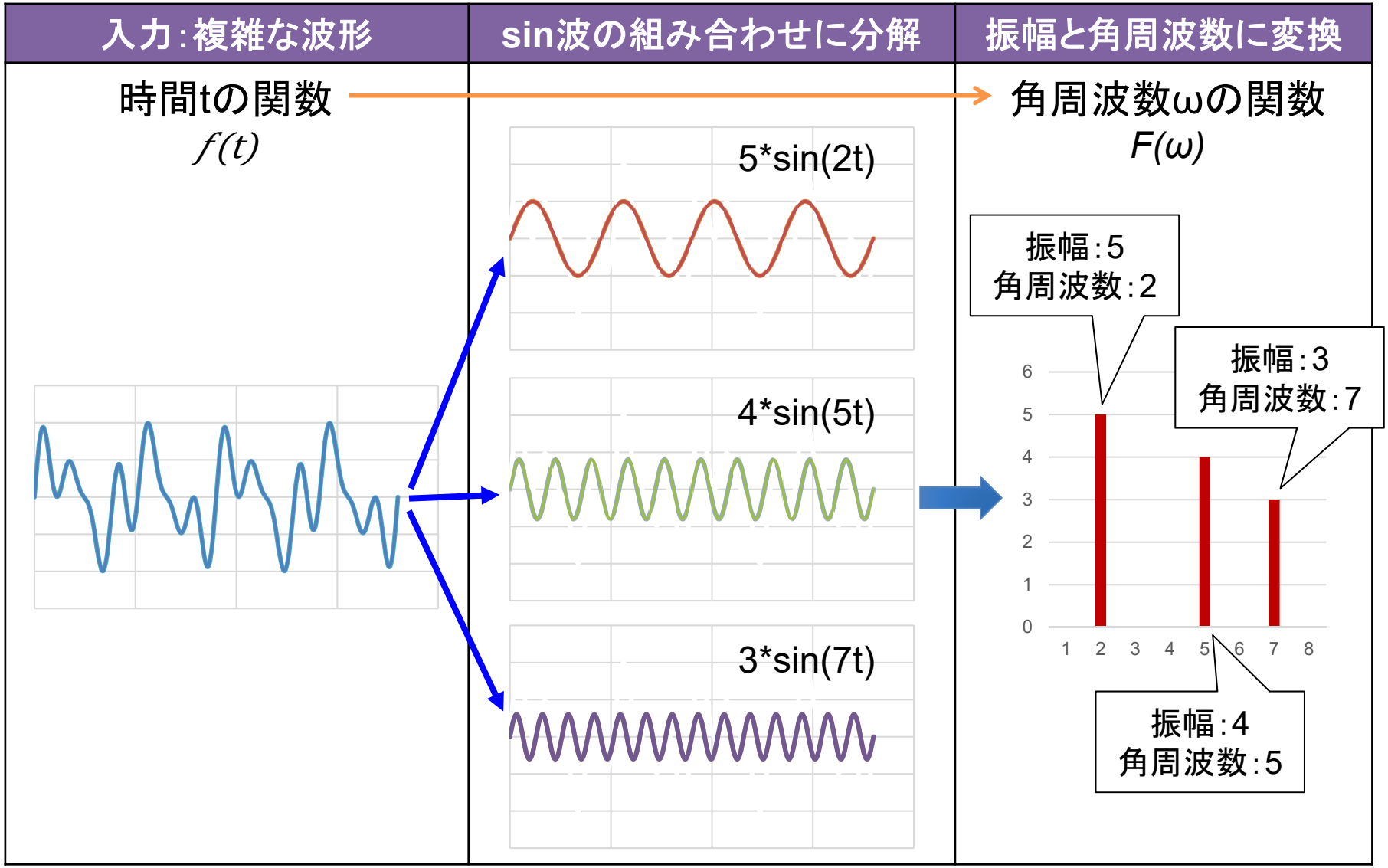


Seiichiro Tani, Hirotada Kobayashi, and Keiji Matsumoto
Exact Quantum Algorithms for the Leader Election Problem
[v1] Thu, 27 Dec 2007
https://arxiv.org/abs/0712.4213

2-4: 量子フーリエ変換

アナログ計算であるフーリエ変換は量子計算と相性が良いアルゴリズムです。また可逆性よりフーリエ変換が可能なら逆フーリエ変換の量子回路も可能となります。

フーリエ変換（波形を振幅と角周波数に分解）



アダマールは 1量子ビット フーリエ変換

※ アダマール変換は2回適用すると元に戻るが位相が逆の逆フーリエ変換でもある。

Step1: アダマール変換を量子フーリエ変換的に解く。

$$\mathrm{H}|x\rangle = \frac{1}{\sqrt{2}}|0\rangle + (-1)^x \frac{1}{\sqrt{2}}|1\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + (-1)^x|1\rangle\right)$$

$$= \frac{1}{\sqrt{2}} \sum_{y=0}^{1} (-1)^{x \cdot y} |y\rangle$$

$x = 0 : \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$
$x = 1 : \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

Step2: N（2^n）数の量子フーリエ変換（$\mathrm{QFT_N}$）に拡張する。

$$\mathrm{QFT_N}\,|x\rangle = \frac{1}{\sqrt{\mathrm{N}}} \sum_{y=0}^{\mathrm{N}-1} \omega^{x \cdot y} |y\rangle$$

$$\omega = \exp\left(\frac{2\pi \mathrm{i}}{N}\right)$$

ωの例：アダマールH（N=2）の時 $\omega = \exp(\pi i) = 180°$

※ 組み合わせ数（$N=2^{n\text{ビット}}$）が増えると角度（位相）は半分ずつ減って行く。

制御位相回転ゲート cRn

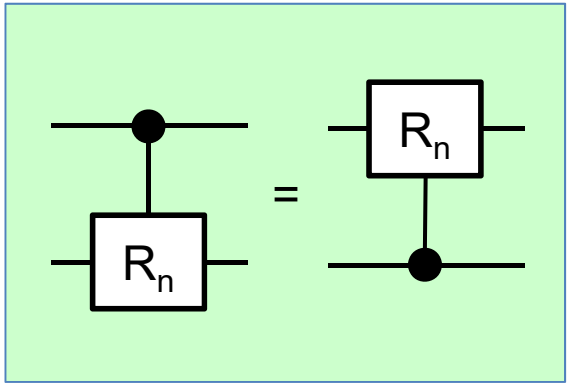
量子フーリエ変換は位相 $\exp\left(\frac{2\pi i}{2^n}\right)$ 計算が必要。

制御付きの位相回転ゲートRx の $x = 2^n$ として、

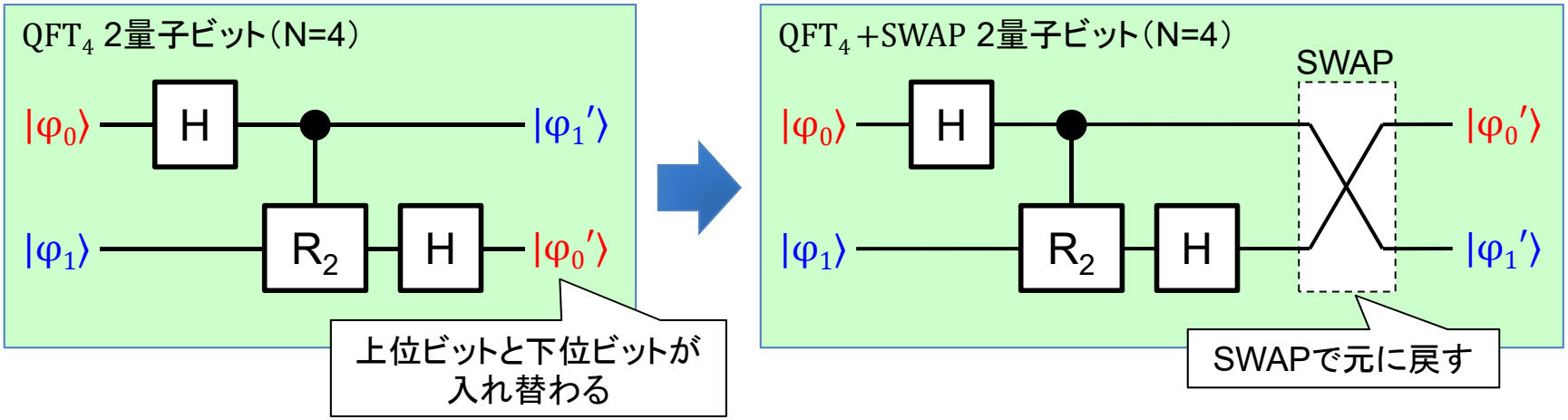
$cR_1 = CZ$, $cR_2 = CZ^{0.5}$, $cR_3 = CZ^{0.25}$,… $cR_n = CZ^{1/2^{n-1}}$

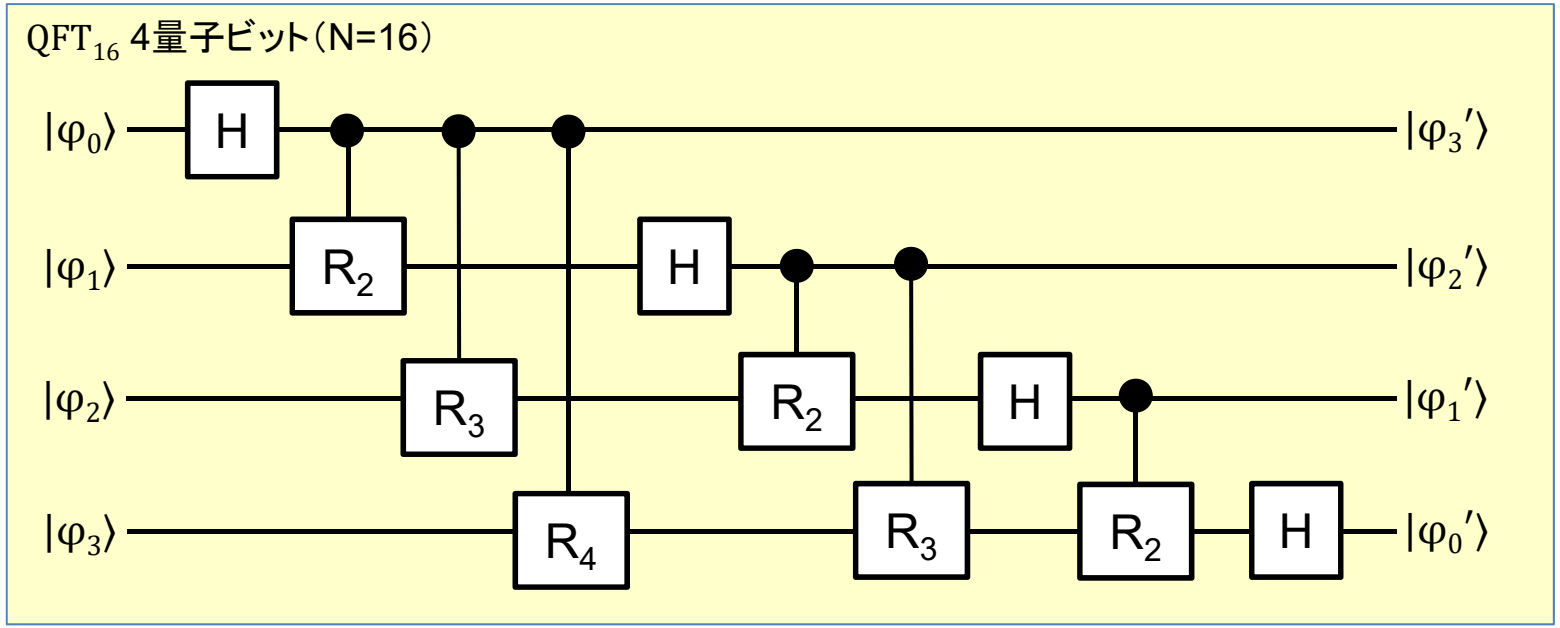
となる。Cirqでは $cR_2 = CZ^{0.5} = CZ(q)**0.5$ とする。

$$cRn = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & (-1)^{1/2^{n-1}} \end{pmatrix} = CZ^{1/2^{n-1}}$$



n量子ビットの量子フーリエ変換 QFT_N





2量子ビットの量子フーリエ変換 計算

1. 入力が振幅(入力ベクトル)なら位相に変換して出力

$$\mathrm{QFT}_4 |00\rangle = \frac{1}{\sqrt{4}} \sum_{y=0}^{3} \omega^{0 \cdot y} |y\rangle$$

$$= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \mathrm{H} |00\rangle$$

2. 入力が位相なら振幅(入力ベクトル)に変換して出力

$$\mathrm{QFT}_4 \, \mathrm{H} |00\rangle = \frac{1}{4} \left(\sum_{y=0}^{3} \left(\sqrt{i}\right)^{0 \cdot y} |y\rangle + \sum_{y=0}^{3} \left(\sqrt{i}\right)^{1 \cdot y} |y\rangle \right.$$

$$\left. + \sum_{y=0}^{3} \left(\sqrt{i}\right)^{2 \cdot y} |y\rangle + \sum_{y=0}^{3} \left(\sqrt{i}\right)^{3 \cdot y} |y\rangle \right)$$

$$= |00\rangle$$

量子干渉効果で打ち消し

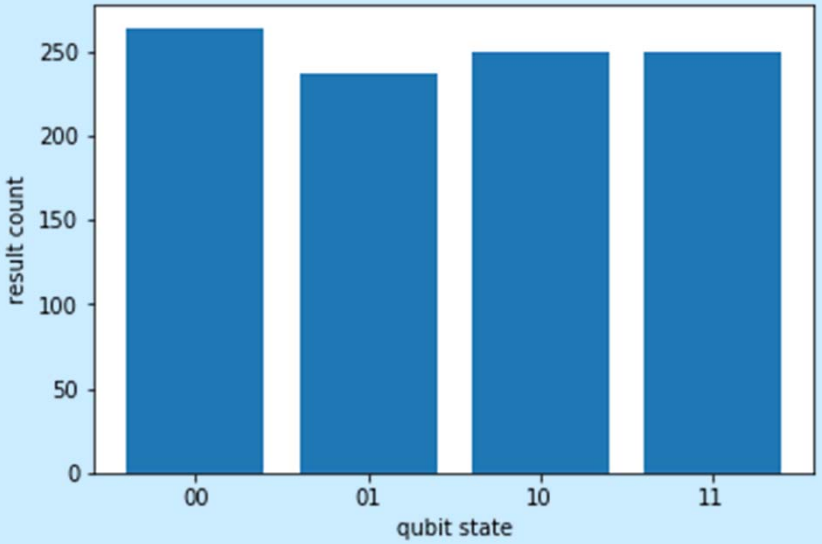
※ $|00\rangle$ を使った場合は、2量子ビットのアダマール変換と同じ。

2量子ビットの量子フーリエ変換 実行1

```
from cirq import *
Q = [LineQubit(i) for i in range(2)]
qc = Circuit()
# 量子フーリエ変換
qc.append(H(Q[0]))
qc.append(CZ(*Q)**0.5)
qc.append(H(Q[1]))
# 上位と下位のビット入れ替え
qc.append(SWAP(Q[0],Q[1]))
# 観測（結果取得）
qc.append(measure(Q[0], key='q0'))
qc.append(measure(Q[1], key='q1'))
print("Circuit:")
print(qc)
# 実行と結果表示
r = Simulator().run(qc, repetitions=1000)
plot_state_histogram(r)
```

```
Circuit:
0: ─H─@─────────×─M('q0')─
      │         │
1: ───@^0.5─H─×─M('q1')─
```

実行結果



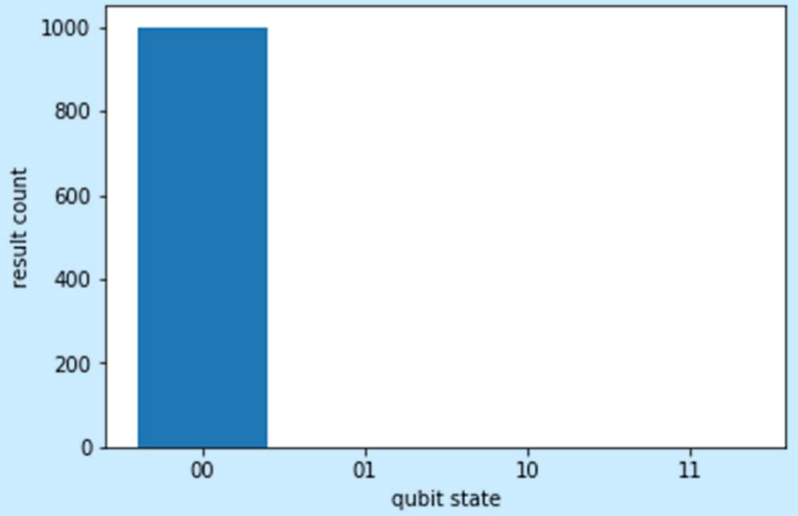
array([264., 237., 250., 249.])

振幅（値）を
位相に変換

2量子ビットの量子フーリエ変換 実行2

```
from cirq import *
Q = [LineQubit(i) for i in range(2)]
qc = Circuit()
# 重ね合わせ状態を入力
qc.append(H.on_each(*Q))
# 量子フーリエ変換
qc.append(H(Q[0]))
qc.append(CZ(*Q)**0.5)
qc.append(H(Q[1]))
# 上位と下位のビット入れ替え
qc.append(SWAP(Q[0],Q[1]))
# 観測（結果取得）
qc.append(measure(Q[0], key='q0'))
qc.append(measure(Q[1], key='q1'))
# 実行と結果表示
r = Simulator().run(qc, repetitions=1000)
plot_state_histogram(r)
```

実行結果



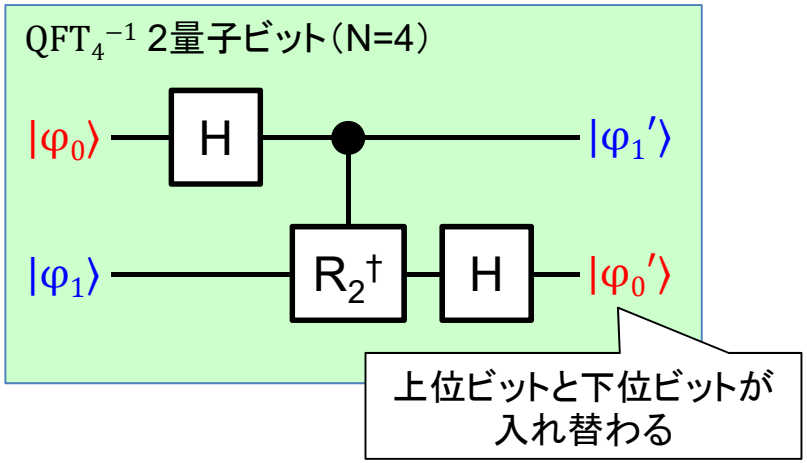
array([1000., 0., 0., 0.])

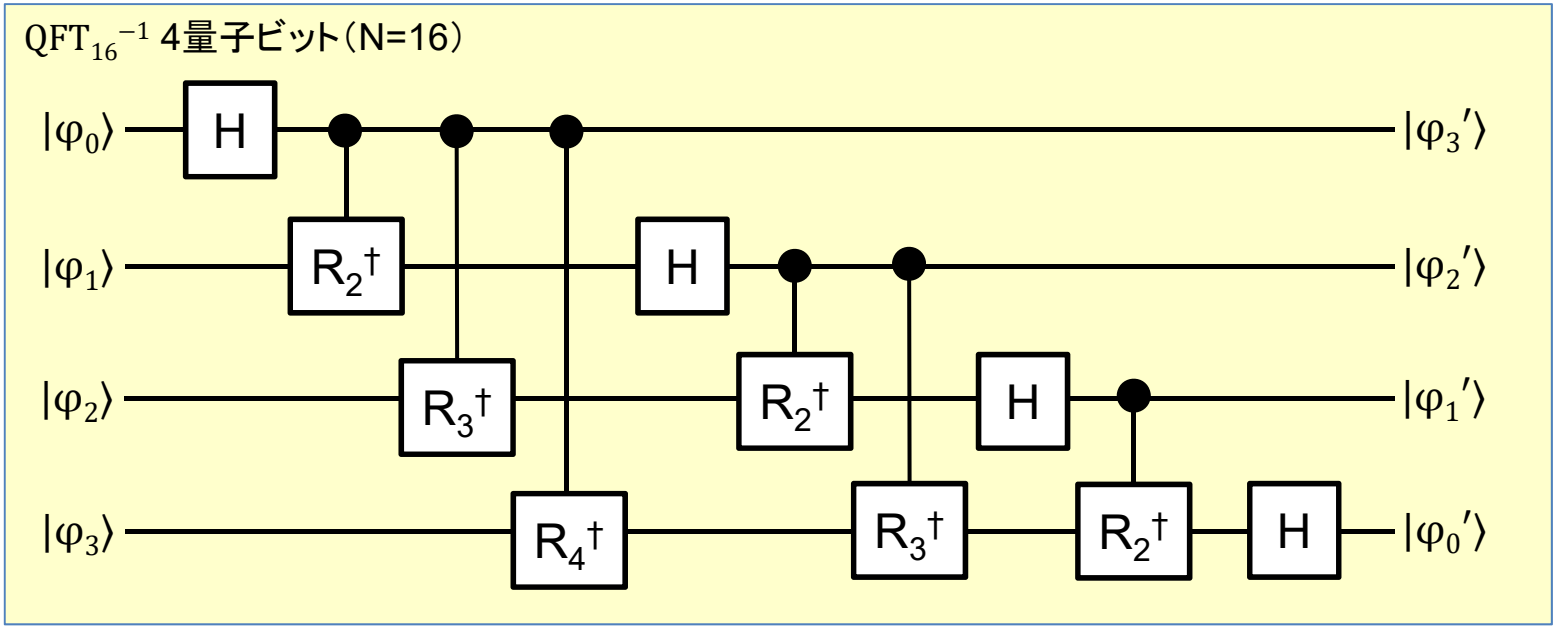
位相を振幅（値）に変換

- 量子干渉効果により重ね合わせ状態が打ち消されている。よく利用される変換。

n量子ビットの逆量子フーリエ変換 QFT_N^{-1}

逆量子フーリエ変換は
位相の向きを逆にする。
R_n を $R_n^{\dagger}$ に変更する。
※ アダマールは反転だった。

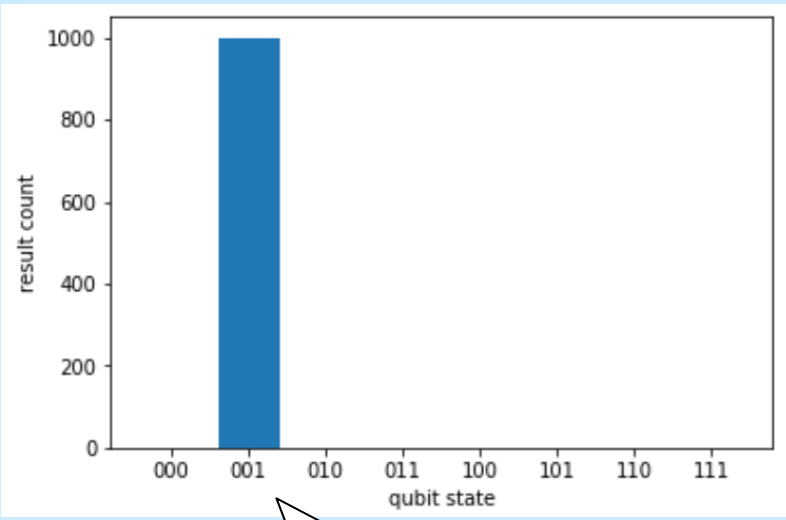




量子フーリエ変換・逆量子フーリエ変換

```
from cirq import *
# 量子フーリエ変換 定義 (inv=-1なら逆変換)
def qft(Q,n,inv):
    for i in range(n):
        for j in range(i):
            yield CZ(Q[i],Q[j])**(inv*1/2**(i-j))
        yield H(Q[i])
# 前準備
n = 3     # 3量子ビット(qbit)
Q = [LineQubit(i) for i in range(n)]
qc = Circuit()
# 入力として最後の量子ビットを反転して |001> に
qc.append(X(Q[2]))
# 量子フーリエ変換 inv = 1
qc.append(qft(Q,n,1))
qc.append(SWAP(Q[0],Q[2]))  # Q[1]は入れ替え不要
# 逆量子フーリエ変換 inv = -1
qc.append(qft(Q,n,-1))
qc.append(SWAP(Q[0],Q[2]))  # Q[1]は入れ替え不要
# 観測 (結果取得)
qc.append(measure(Q[0], key='q0'))
qc.append(measure(Q[1], key='q1'))
qc.append(measure(Q[2], key='q2'))
# 実行と結果表示
r = Simulator().run(qc, repetitions=1000)
plot_state_histogram(r)
print(r)
```

実行結果



初期値の
|001⟩ に
戻っている

2-5: ショアのアルゴリズム

素因数分解を行うショアのアルゴリズムにより
公開鍵暗号が破れると言う注目をされました。

実際には必要なスケールを持つ量子ハードを
実現できないのですが正しく恐れる為に、
アルゴリズムを学びましょう。

素因数分解問題

ある正の整数を、素数の積の形で表わすこと。
ここでは正の整数の入力から、2つの素因数の積に分解する(2つの異なる素因数を得る)問題と考える。

※ 素因数:整数の因数である約数のうち素数であるもの。

例1:15 = 3 × 5　　(入力:15、出力:3と5)
例2:221 = 13 × 17　　(入力:221、出力:13と17)
例3:21631 = 97 ×223 (入力:21631、出力:97と223)

古典計算にて安直に解くには順番に素数を掛けて試して行く。
例2なら、221÷2、221÷3、221÷5、...、221÷13=17(正解!)とすることで解けるが元の整数が大きくなると指数的に困難になる。

※ RSA暗号は素因数分解が困難であることを前提とした暗号方式である。

ショア(Shor)のアルゴリズム

1994年にIBMのピーター・ショアが発表した、**素因数分解アルゴリズム**。(25年前なので結構古い)

全てを量子計算する訳では無く、位数発見部を量子計算する(他は古典計算する)ことで、多項式時間(n個の計算を n^c 回で計算、c は定数)で計算可能とする。

※ 古典解法では指数時間(n個の計算を c^n 回で計算)が必要だった。

量子と古典の両方の計算を使う、**ハイブリッド計算**が必要になるのでPythonによる計算は適している。

※ 素因数分解は可能だがビット数が増えると必要な量子ビット数が増える。この為、**実用にはハードウェアの進歩が必要。**

ショアのアルゴリズムの手順

ショアの計算では以下の3ステップが必要。

Step1：前処理（古典計算）

- 入力された整数Nから任意の数aを選択。
- Nとaは互いに素である必要がある。

Step2：周期発見（量子計算）

- 数aを使って量子的に位数発見問題を解く。
- 位相の数から位数 r（周期T）を得る。

Step3：後処理（古典計算）

- 位数 r をチェックして正しくなければStep1へ。
  位数 r を使い2つの素因数を計算して終了。

フェルマー(Fermat)テスト(確率的素数判定)

古典計算

互いに素:

2つの整数の最大公約数が1である。
2つの整数 a, b の**最大公約数 gcd(a, b) が 1** と同じ。

フェルマーの小定理:

pが素数の時、pと互いに素な整数aに以下が成り立つ。

$a^p \equiv a \pmod p \Rightarrow a^{p-1} \equiv 1 \pmod p$

フェルマーテスト(素数判定):

aとpが互いに素の時。
フェルマーの小定理の対偶を使うと以下が言える。

$a^{p-1} \not\equiv 1 \pmod p$ なら、**pは素数ではない。**

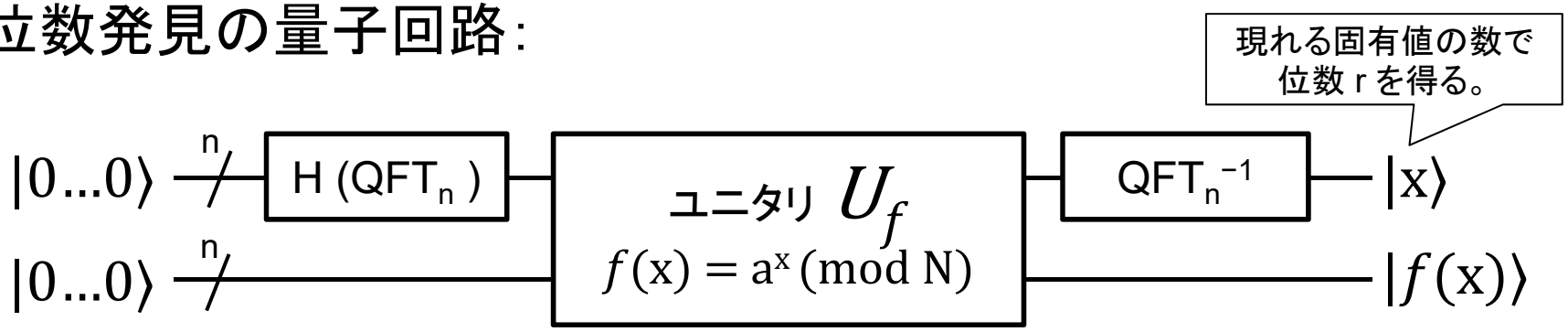
位数(周期)発見問題

関数 $f(x) = a^x \pmod N$ において、
$f(x) = f(x+r)$ となる位数 r を発見する。

※ 位相が分かればその数から位数が高確率で取得できる。

⇒ (逆)量子フーリエ変換を使い、位数を発見する事が可能。

位数発見の量子回路:



※ 量子フーリエ変換は全ビットをアダマール変換すれば良く、逆量子フーリエ変換は既出である。

関数 $f(x) = a^x \pmod N$ のユニタリ回路が必要。

位数(周期)発見の回路：$a^x \pmod N$

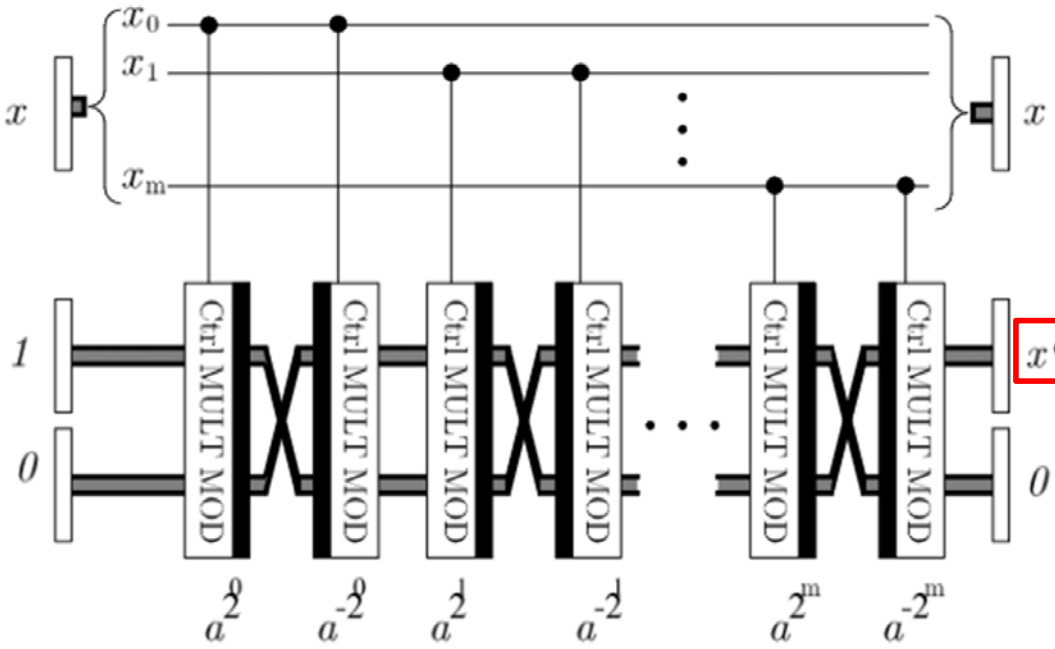
入力：整数 N と任意の数 a を入力。← 2入力
出力：位数 r を得る。← 1出力
実現：位数発見の汎用的モジュロ回路が必要！

推薦資料「Shorのアルゴリズム」
加藤 拓己*, 湊 雄一郎*, 中田 真秀**（*MDR株式会社, **理化学研究所）
https://speakerdeck.com/gyudon/shorfalsearugorizumu

参考資料「量子計算機の到来を正しく恐れたい」
國廣 昇（東京大学）：NICTサイバーセキュリティシンポジウム2019発表資料
※ 本資料の疑問への答えが、上記の「Shorのアルゴリズム」と言える。
https://www2.nict.go.jp/csri/plan/H31-symposium/pdf/kunihiro.pdf

以上の資料をベースにまとめる。

$x^a \bmod N$ を計算する量子ゲート回路



入力ビット数(L)に対し:
- 量子ビット数 → 3L+2 qbit
  L=2048bit なら 6146 qbit
- 量子ゲート数 → L^3 ゲート
  L=2048bit なら $2048^3 = 10^9$

量子ビット数を減らす場合:
- 量子ビット数 → 2L+2 qbit
  L=2048bit なら 4098 qbit
- 量子ゲート数 → L^4 ゲート
  L=2048bit なら $2048^4 = 10^{13}$

Figure 6)

V. Vedral, A. Barenco and A. Ekert

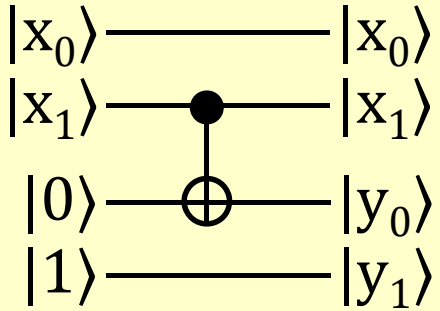
V. Vedral, A. Barenco, A. Ekert,
Quantum Networks for Elementary Arithmetic Operation,
arXiv:quant-ph/9511018 (1995)
https://arxiv.org/abs/quant-ph/9511018

「いちばんやさしい量子コンピュータで暗号を解くshorのアルゴリズム概要」
Blueqatを使ったモジュロ演算の実装例　@YuichiroMinato（MDR）
https://qiita.com/YuichiroMinato/items/5f98c467c006d7cb902c

簡易:位数発見問題のユニタリ回路1

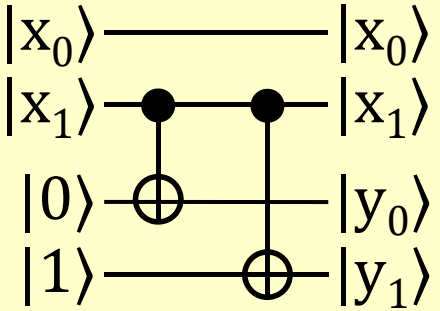
関数 $y = f(x) = a^x \bmod N$ (x=0,1,2,...) のユニタリ回路が必要。
まともにやるのは大変なので真理値表から作って試験する。

$f(x) = 3^x \pmod 8$



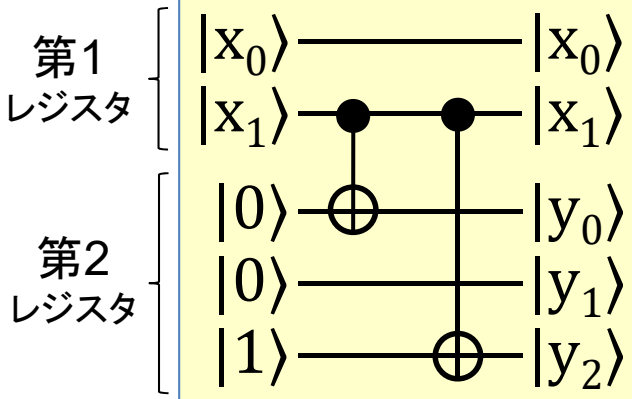
x_0	x_1	y_0	y_1
0	0	0	1
0	1	1	1
1	0	0	1
1	1	1	1

$f(x) = 2^x \pmod 3$



x_0	x_1	y_0	y_1
0	0	0	1
0	1	1	0
1	0	0	1
1	1	1	0

$f(x) = 4^x \pmod{15}$



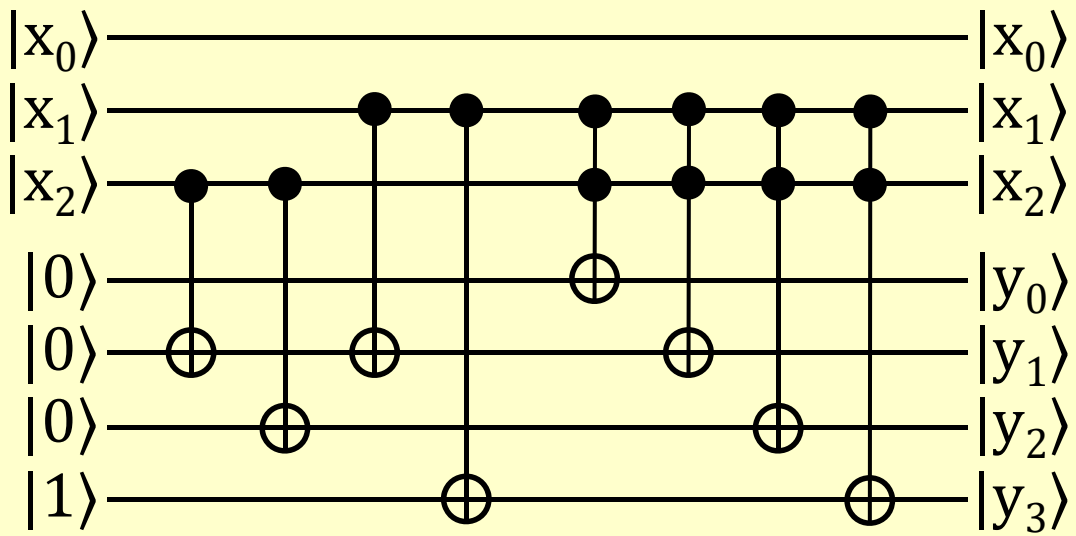
x_0	x_1	y_0	y_1	y_2
0	0	0	0	1
0	1	1	0	0
1	0	0	0	1
1	1	1	0	0

この方法では位数rを発見する回路の為に全ケースを先に計算(位数rは明確)をしているので実用的では無いが最小量子ビット数で回路を実現できるので学習用として利用されている。

簡易:位数発見問題のユニタリ回路2

でも $f(x) = 7^x \pmod{15}$ だとこんなに複雑に…

$f(x) = 7^x \pmod{15}$



x_0	x_1	x_1	y_0	y_1	y_2	y_3
0	0	0	0	0	0	1
0	0	1	0	1	1	1
0	1	0	0	1	0	0
0	1	1	1	1	0	1
1	0	0	0	0	0	1
1	0	1	0	1	1	1
1	1	0	0	1	0	0
1	1	1	1	1	0	1

この程度でも汎用的な位数発見回路が必要と思える。
しかしまともにやると必要となる量子ビット数が増える…

Step1：前処理（古典計算）

入力値NからNより小さく互いに素な因数aを選択する。

Nと互いに素な素因数aの見つけ方：

1. Nより小さい整数aを選ぶ。
2. Nとaの最大公約数 gcd(N,a) が1ならaは素因数（※）。
3. 最大公約数が1以外なら別の整数aを選びやり直す。

※「ユークリッドの互除法」により最大公約数が1なら「互いに素」となる。

例：入力値N＝15が与えられた場合

解：互いに素な値 a は、2, 4, 7, 8, 11, 13, 14 となる。

```
import math              # gcdを使う為にmathを利用
N = 15                   # 入力値N
for i in range(2,N):     # 1は除くので2からNまで
    r = math.gcd(N,i)    # 最大公約数の計算
    if r == 1:           # rが1なら互いに素
        print(i)         # 互いに素な値a表示
```

古典計算

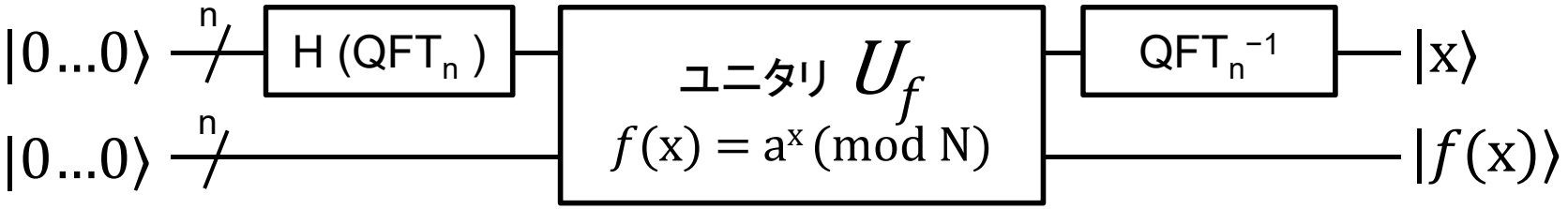
実行

```
2
4
7
8
11
13
14
```

Step2：周期発見（量子計算）

入力値NとStep1で計算した値aから位数を計算。

適切なユニタリ回路を組み、位数発見問題を解く。



例：入力値N＝15の場合に、a＝4と7を試してみる。

解：次ページ以降に記載の量子回路を実行する。

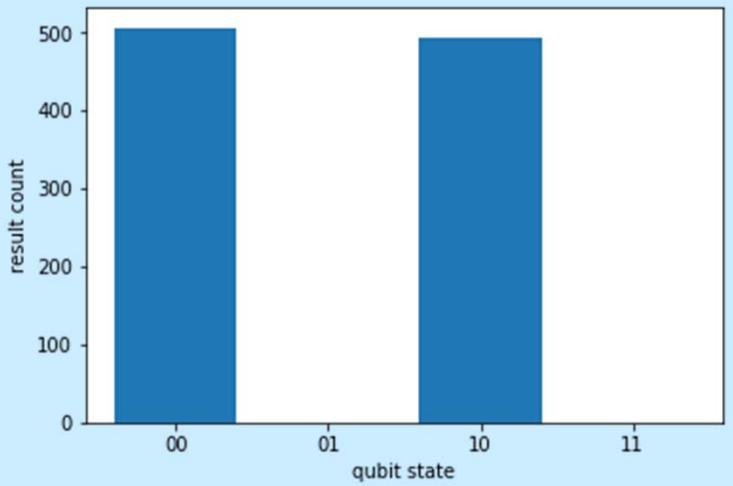
N＝15, a＝4 の場合の位数 r ＝ 2 を得る

N＝15, a＝7 の場合の位数 r ＝ 4 を得る

周期発見 N=15, a=4 の実行

```
from cirq import *
# 量子フーリエ変換 定義 (inv=-1なら逆変換)
def qft(Q,n,inv):
    for i in range(n):
        for j in range(i):
            yield CZ(Q[i],Q[j])**(inv*1/2**(i-j))
        yield H(Q[i])
# 前準備
n = 3        # yは3qbit (入力N=15なので充分)
Qx = [GridQubit(0,i) for i in range(n-1)] # xは2qbit
Qy = [GridQubit(1,i) for i in range(n)]
qc = Circuit()
# 計算用Qxを重ね合わせ状態にする
qc.append(H.on_each(*Qx))
# オラクル計算
qc.append(X(Qy[n-1]))
qc.append(CNOT(Qx[1], Qy[0]))
qc.append(CNOT(Qx[1], Qy[2]))
# Qxを逆量子フーリエ変換 inv = -1
qc.append(qft(Qx,n-1,-1))
qc.append(SWAP(Qx[0],Qx[1]))
# Qxの観測 (結果取得)
qc.append(measure(Qx[0], key='q0'))
qc.append(measure(Qx[1], key='q1'))
# 実行と結果表示
r = Simulator().run(qc, repetitions=1000)
plot_state_histogram(r)
```

実行結果



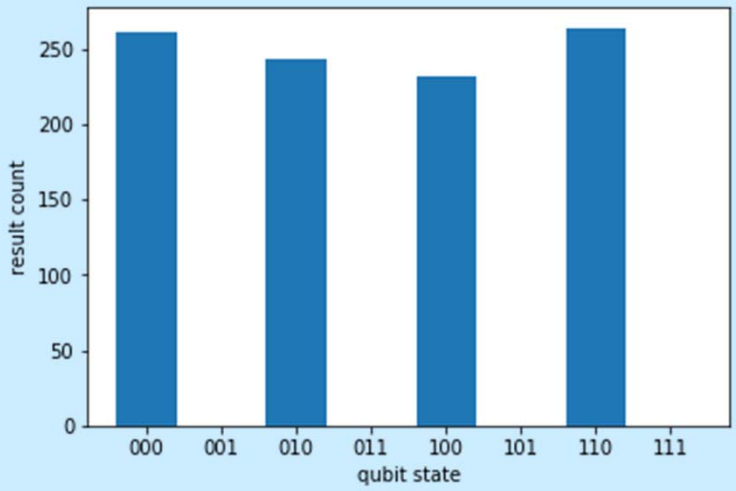
array([506., 0., 494., 0.])

値が出るのは、
00, 10 の2通りなので
r=2

周期発見 N=15, a=7 の実行

```
from cirq import *
# 量子フーリエ変換 定義 (inv=-1なら逆変換)
def qft(Q,n,inv):
    for i in range(n):
        for j in range(i):
            yield CZ(Q[i],Q[j])**(inv*1/2**(i-j))
        yield H(Q[i])
# 前準備
n = 4        # yは4qbit (入力N=15なので充分)
Qx = [GridQubit(0,i) for i in range(n-1)] # xは3qbit
Qy = [GridQubit(1,i) for i in range(n)]
qc = Circuit()
# 計算用Qxを重ね合わせ状態にする
qc.append(H.on_each(Qx))
# オラクル計算
qc.append(X(Qy[n-1]))
qc.append(CNOT(Qx[2], Qy[1]))
qc.append(CNOT(Qx[2], Qy[2]))
qc.append(CNOT(Qx[1], Qy[1]))
qc.append(CNOT(Qx[1], Qy[3]))
qc.append(CCX(Qx[1], Qx[2], Qy[0]))
qc.append(CCX(Qx[1], Qx[2], Qy[1]))
qc.append(CCX(Qx[1], Qx[2], Qy[2]))
qc.append(CCX(Qx[1], Qx[2], Qy[3]))
# Qxを逆量子フーリエ変換 inv = -1
qc.append(qft(Qx,n-1,-1))
qc.append(SWAP(Qx[0],Qx[2]))
# Qxの観測 (結果取得)
qc.append(measure(Qx[0], key='q0'))
qc.append(measure(Qx[1], key='q1'))
qc.append(measure(Qx[2], key='q2'))
# 実行と結果表示
r = Simulator().run(qc, repetitions=1000)
plot_state_histogram(r)
```

実行結果



array([261., 0., 243., 0., 232., 0., 264., 0.])

値が出るのは、
000, 010, 100, 110
の4通りなので r=4

Step3：後処理（古典計算）

Step2で計算された位数 r を使って以下をチェック。

1. 位数rが奇数ならば、Step1から別のaでやり直し。
2. 位数rが偶数かつ、$a^{r/2}\pm1$とNが互いに素であれば、Step1から別のaでやり直し。
3. 周期Tが偶数で、$a^{r/2}\pm1$とNが互いに素で無ければ、以下の計算で2つの素因数が得られる。

$$\text{素因数 P1} = \gcd(a^{r/2}+1, N)$$
$$\text{素因数 P2} = \gcd(a^{r/2}-1, N)$$

例：N=15, a=7, r=4 の時、

$$\text{素因数 P1} = \gcd(7^2+1, 15) = \gcd(50, 15) = 5$$
$$\text{素因数 P2} = \gcd(7^2-1, 15) = \gcd(48, 15) = 3$$

解：15 の素因数は 5 と 3 である（5×3=15 なので正しい）。

N=15, a=2,4,7,8,11,13,14 の周期表

関数 $f(x) = a^x \bmod 15$ の計算結果：

$a^{r/2} \pm 1$とNが互いに素ならNGとする。

a	^1	^2	^3	^4	^5	^6	^7	^8	^9	^10	^11	^12	r	chk
2	2	4	8	1	2	4	8	1	2	4	8	1	4	OK
4	4	1	4	1	4	1	4	1	4	1	4	1	2	OK
7	7	4	13	1	7	4	13	1	7	4	13	1	4	OK
8	8	4	2	1	8	4	2	1	8	4	2	1	4	OK
11	11	1	11	1	11	1	11	1	11	1	11	1	2	NG
13	13	4	7	1	13	4	7	1	13	4	7	1	4	OK
14	14	1	14	1	14	1	14	1	14	1	14	1	2	NG

a=2, r=4 : $P1 = \gcd(2^2+1, 15) = \gcd(5, 15) = 5$ / $P2 = \gcd(2^2-1, 15) = \gcd(3, 15) = 3$

a=4, r=2 : $P1 = \gcd(4^1+1, 15) = \gcd(5, 15) = 5$ / $P2 = \gcd(4^1-1, 15) = \gcd(3, 15) = 3$

a=7, r=4 : $P1 = \gcd(7^2+1, 15) = \gcd(50, 15) = 5$ / $P2 = \gcd(7^2-1, 15) = \gcd(48, 15) = 3$

a=8, r=4 : $P1 = \gcd(8^2+1, 15) = \gcd(65, 15) = 5$ / $P2 = \gcd(8^2-1, 15) = \gcd(63, 15) = 3$

a=11, r=2 : $P1 = \gcd(11^1+1, 15) = \gcd(12, 15) = 12$ / $P2 = \gcd(11^1-1, 15) = \gcd(10, 15) = 10$

a=13, r=4 : $P1 = \gcd(13^2+1, 15) = \gcd(170, 15) = 5$ / $P2 = \gcd(13^2-1, 15) = \gcd(168, 15) = 3$

a=14, r=2 : $P1 = \gcd(14^1+1, 15) = \gcd(15, 15) = 0$ / $P2 = \gcd(14^1-1, 15) = \gcd(13, 15) = 13$

Step3の補足説明

$f(x) = f(x+r)$ から、
$a^x \bmod N = a^{x+r} \bmod N$ となる。
これより $a^r \bmod N = 1$ が導かれる。

※ r が偶数ならユークリッドの互除法により素因数が見つかる。

$a^r \bmod N = 1$ で位数 r が偶数なら、
$(a^{r/2}-1)(a^{r/2}+1) \bmod N = 0$ となり、
最大公約数 $\gcd(a^{r/2}\pm 1, N)$ により、
因数が高い確率(50%以上)で計算できる。

現在のRSA暗号は解けるのか？

※ 現在2048～4096ビットのRSA暗号が使われている。

2048ビット（L=2048）を解こうとした場合：

- 量子ビット数：$2048 \times 3 + 2 = 6146$（エラー無し）
- 量子ゲート数：$2048^3 = 86$億（持続時間に影響）

が必要となる（量子ゲート数は古典だとステップ数に相当）。

エラー無しの6148量子ビットの実現：

- 量子ビット数もだがエラー無しハードルは高い。

86億ゲートを実現する重ね合わせ持続時間の実現：

- 量子重ね合わせ状態維持もハードルが高い。

正直言って実現することはかなり難しい？

2-6: エラー訂正問題

量子ゲート型の量子コンピュータの課題は、
量子ビット数を増やすこととエラー訂正である。

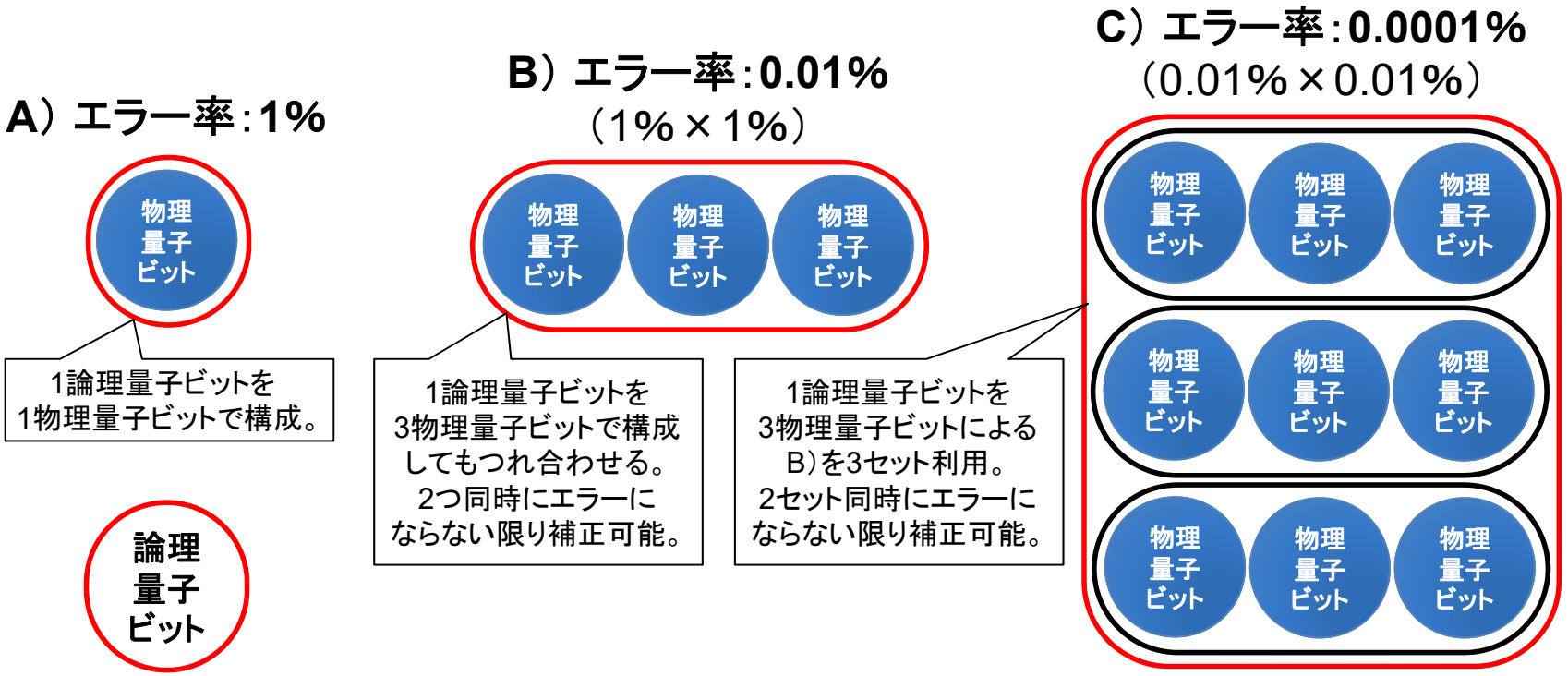
量子ビットエラー訂正の難しさ

古典計算での1ビットエラーは反転しかない。
0→1 になるか、1→0 になるかのどちらかのみ。
※もちろん古典計算でも複数ビットへの影響の可能性はある。

- **量子計算1ビットエラーはX/Y/Zの3種類がある。**
  - ビット反転 X(q) エラー
  - 位相ビット反転 Y(q) エラー
  - 位相反転 Z(q) エラー
- **量子計算の途中でエラー訂正する必要がある。**
  - 重ね合わせ状態の量子ビットを測定することなく訂正する。
  - ショアのエラー訂正アルゴリズム以前は不可能と言われた。

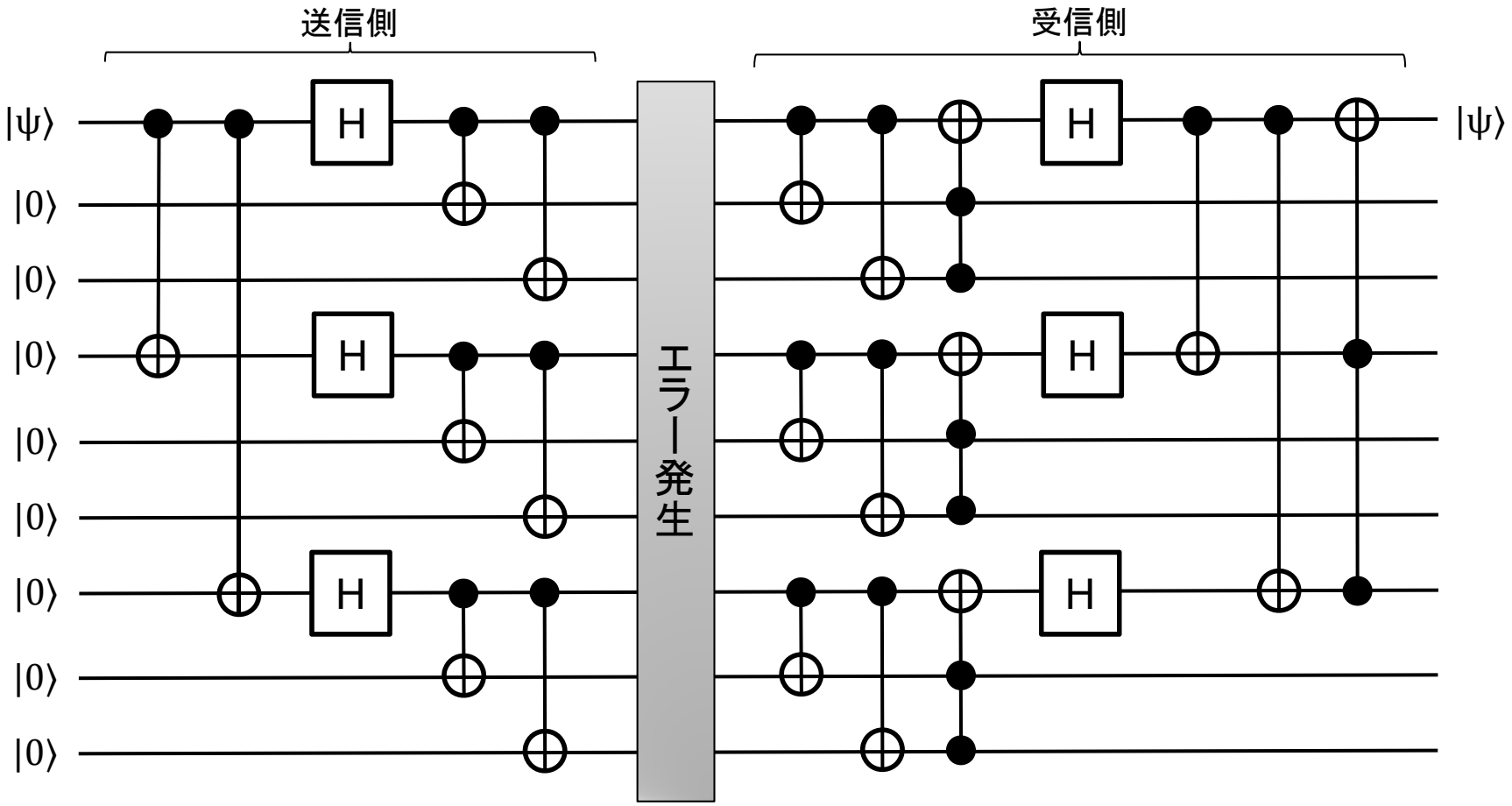
物理量子ビットと論理量子ビット

必要なのは量子計算に利用するエラー訂正された
論理量子ビットである。**論理**量子ビットは複数の
物理量子ビットを組み合わせてエラー訂正して実現。



量子ビット(ショア)のエラー訂正回路

ショアは量子パリティビットを導入して解決の道を示したが、
必要となる量子ビット数は増大する(この場合9量子ビットが必要)。



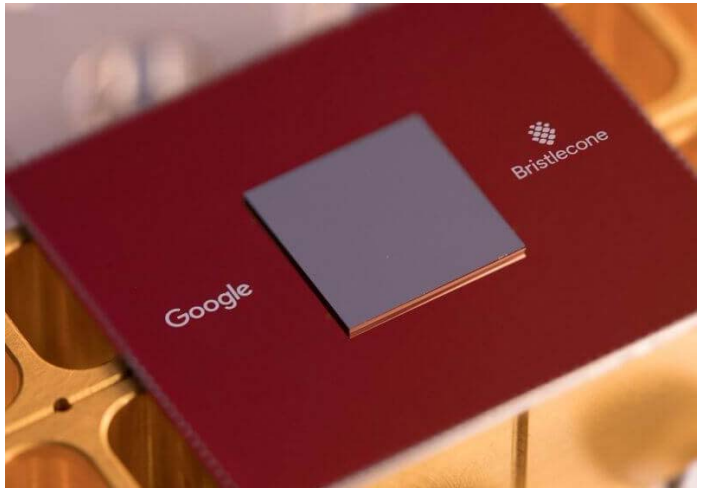
2-7：Cirq（Google）・Blueqat（MDR）

本資料ではGoogleのCirqを使って来たが少し
Cirqの補足説明をする。

また日本発の量子計算フレームワークとして
Blueqatが公開されており、簡単に紹介する。

Google 量子コンピュータへの取り組み

Googleは、2018年3月5日、72量子ビット量子プロセッサ Bristlecone を発表。



2018年7月18日に、NISQ用フレームワーク Cirq を公開。

Bristleconeは極低温の超伝導状態で動作するタイプの量子計算用チップ。Googleは1チップにまとめることで低エラーの達成と実用を目指す。

Bristlecone以外にもイオントラップ方式のIonQにも投資中（Amazonも投資）で、色々手を打っている。

cirq.Simulator の run と simulate

- cirq.Simulator.simulate(qc)
  - 重ね合わせ状態のまま、値を取得できる量子シミュレータ。
  - ノイズ無しの理論値（確率振幅）を取得できる。
  - 計算途中の重ね合わせ状態の確認時に利用する。
- cirq.Simulator.run(qc,...)
  - ノイズあり（NISQ用）の量子シミュレータ。
  - 観測しないと結果の取得はできない。

※ cirq.google.Bristlecone
  - おそらく実機 Bristlecone 実行用。
  - 現時点では未サポート。

Blueqatを使う（MDR）

環境：**Anaconda3**（**Python3.5**）

以下より環境に合わせてダウンロードとインストール

https://www.anaconda.com/distribution/

ライブラリ：**Blueqat**（ブルーキャット）

Windows版：Anaconda Prompt

MacOS版：ターミナル

インストール

```
pip install blueqat
```

バージョン指定インストール

```
pip install blueqat=0.3.9
```

アンインストール

```
pip uninstall blueqat
```

※ Blueqatのバージョン確認：

In:	import blueqat blueqat.__version__
Out:	'0.3.9‘

本資料のソースは 0.3.9 と表示される環境にて確認しています。

Blueqat 超入門（量子 Hello World!）

アダマールゲートHの実行

```
from blueqat import *
Circuit().h[0].m[:].run(shots=1000)
```

量子回路生成　アダマール　観測　実行（1000ショット）

同じ

実行結果:

```
Counter({'1': 488, '0': 512})
```

```
from blueqat import *
qc = Circuit(1)   # 1qbit指定
qc.h[0].m[:]      # 回路
r = qc.run(shots=1000) # 実行
print(r)          # 表示
```

複数ビット操作:

```
Circuit().z[1:3] # Z on 1,2
Circuit().x[:3]  # X on (0, 1, 2)
Circuit().h[:]   # H on all qubits
Circuit().x[1,2] # 1qubit gate
```

ローテーション操作:

```
Circuit().rz(math.pi/4)[0] # Z rotate pi/4
```

短い!

Blueqat 便利機能:ユニタリ行列表示

回路(ビット反転)指定してユニタリ行列を表示:

```
from blueqat import *
Circuit().x[0].run(backend="sympy_unitary")
```

実行結果(x[0]):

```
Matrix([
[0, 1],
[1, 0]])
```

$$= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

実行結果(アダマール h[0]):

```
Matrix([
[1/sqrt(2),  1/sqrt(2)],
[1/sqrt(2), -sqrt(2)/2]])
```

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

表現が違うが値は正しい

回路(複数ゲート)指定してユニタリ行列を表示:

```
from blueqat import *
Circuit().h[0].x[0].h[0].run(backend="sympy_unitary")
```

実行結果(HXHなのでZゲートと同じ):

```
Matrix([
[1,  0],
[0, -1]])
```

$$= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

複数ゲートでも
使えるので便利。

Blueqatの情報

開発者が日本人なので日本語の情報が多い。

ソースリポジトリ:
https://github.com/Blueqat/

オンライン・ドキュメント(日本語):
https://blueqat.readthedocs.io/ja/latest/

ブログ情報(日本語):
https://qiita.com/tags/blueqat

MDR株式会社:
https://mdrft.com/?hl=ja

MDR開発中の実機が完成したらBlueqatから使える予定。

付録1：Cirq/Qiskit/Blueqat対応表（基本編）

Gate	Cirq	Qiskit	Blueqat
恒等演算		iden	i
ビット反転演算	X	x	x
位相ビット反転演算	Y	y	y
位相反転演算	Z	z	z
アダマール演算	H	h	h
$\frac{\pi}{4}$位相シフト演算	S	s	s
$\frac{\pi}{8}$位相シフト演算	T	t	t
$-\frac{\pi}{4}$位相シフト演算	inverse(S)	sdg	sdg
$-\frac{\pi}{8}$位相シフト演算	inverse(T)	tdg	tdg
測定	measure	measure	m, measure
制御反転演算	CNOT	cx	cx, cnot
交換演算	SWAP	swap	swap
トフォリ演算	CCX, TOFFOLI	ccx	ccx, toffoli

付録2：Cirq/Qiskit/Blueqat対応表（拡張編）

Gate	Cirq	Qiskit	Blueqat
X軸任意回転演算	Rx	rx	rx
Y軸任意回転演算	Ry	ry	ry
Z軸任意回転演算	Rz	rz	rz
制御位相ビット反転演算		cy	
制御位相反転演算	CZ	cz	cz
トフォリ位相反転演算	CCZ		ccz
制御交換演算	CSWAP	cswap	
指定角λ演算		u1	u1
指定角φ,λ演算		u2	u2
指定角θ,φ,λ演算		u3	u3
制御指定角λ演算		cu1	cu1
制御指定角φ,λ演算		cu2	cu2
制御指定角θ,φ,λ演算		cu3	cu3
量子コンピュータ実機	Google（予定）	IBM	MDR（予定）

Part n: エピローグ

量子ゲート型の状況は…

量子アルゴリズムは現在盛んに論文が出ているので日進月歩の状況です。

本資料もすぐに陳腐化する可能性があります。
新しい手法が開発されていないか確認しましょう。

とは言え位相キックバックやオラクル等のブラックボックスを使った手法は基本となります。

実機によるプログラミングはまだ制限が多いのでしばらくは量子シミュレータを活用しましょう。

次回はアニーリング計算プログラミングに関して学びます。是非次回もご参加ください。

プログラマの為の量子コンピュータ入門 目次:

Part 1: 関連数学と1量子ビット操作

- 線形代数学の基本知識
- ブラケット記法と量子計算
- ブロッホ球と1量子ビット操作（IBM Qiskit）

Part 2: 量子ゲート型のプログラミング

- 2量子ビット以上の量子回路（量子もつれ）
- 量子アルゴリズム（グローバ検索・フーリエ変換等）
- 量子ゲート型プログラミング（Google Cirq）

Part 3: 量子アニーリング型のプログラミング

- イジングモデルとQUBOと量子アニーリング
- 量子アルゴリズム（セールスマン巡回問題）
- 量子アニーリング型プログラミング（Blueqat等）

次回!

http://scienceinoh.jp/schrodinger/